Lecture Notes in Computer Science 7542

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, UK
Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA
Josef Kittler
University of Surrey, Guildford, UK
Jon M. Kleinberg
Cornell University, Ithaca, NY, USA
Alfred Kobsa
University of California, Irvine, CA, USA
Friedemann Mattern
ETH Zurich, Switzerland
John C. Mitchell
Stanford University, CA, USA
Moni Naor
Weizmann Institute of Science, Rehovot, Israel
Oscar Nierstrasz
University of Bern, Switzerland
C. Pandu Rangan
Indian Institute of Technology, Madras, India
Bernhard Steffen
TU Dortmund University, Germany
Madhu Sudan
Microsoft Research, Cambridge, MA, USA
Demetri Terzopoulos
University of California, Los Angeles, CA, USA
Doug Tygar
University of California, Berkeley, CA, USA
Gerhard Weikum
Max Planck Institute for Informatics, Saarbruecken, Germany

Bernhard Beckert Ferruccio Damiani
Frank S. de Boer Marcello M. Bonsangue (Eds.)

Formal Methods for Components and Objects

10th International Symposium, FMCO 2011
Turin, Italy, October 3-5, 2011
Revised Selected Papers

Volume Editors

Bernhard Beckert
Karlsruhe Institute of Technology (KIT)
Institute for Theoretical Informatics
Am Fasanengarten 5, 76131 Karlsruhe, Germany
E-mail: beckert@kit.edu

Ferruccio Damiani
University of Turin
Department of Computer Science
Corso Svizzera 185, 10149 Torino, Italy
E-mail: ferruccio.damiani@di.unito.it

Frank S. de Boer
Centre for Mathematics and Computer Science, CWI
Science Park 123, 1098 XG Amsterdam, The Netherlands
E-mail: f.s.de.boer@cwi.nl

Marcello M. Bonsangue
Leiden University
Leiden Institute of Advanced Computer Science (LIACS)
P.O. Box 9512, 2300 RA Leiden, The Netherlands
E-mail: marcello@liacs.nl

ISSN 0302-9743
e-ISSN 1611-3349
ISBN 978-3-642-35886-9
e-ISBN 978-3-642-35887-6
DOI 10.1007/978-3-642-35887-6
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012954549

CR Subject Classification (1998): D.2.4, D.2, D.3, F.3, D.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

© Springer-Verlag Berlin Heidelberg 2013
This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.
The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Large and complex software systems provide the necessary infrastructure in all industries today. In order to construct such large systems in a systematic manner, the focus in development methodologies has switched in the last two decades from functional issues to structural issues: both data and functions are encapsulated into software units which are integrated into large systems by means of various techniques supporting reusability and modifiability. This encapsulation principle is essential to both the object-oriented and the more recent component-based software engineering paradigms.

Formal methods have been applied successfully to the verification of medium-sized programs in protocol and hardware design. However, their application to the development of large systems requires more emphasis on specification, modeling, and validation techniques supporting the concepts of reusability and modifiability, and their implementation in new extensions of existing programming languages like Java.

The 10th Symposium on Formal Methods for Components and Objects (FMCO 2011) was held at the Natural Science Museum in Turin, Italy, during October 3–5, 2011. FMCO 2011 was realized as a concertation meeting of European projects focussing on formal methods for components and objects. This volume contains 20 revised papers submitted after the symposium by the speakers of each of the following European projects involved in the organization of the program:

- The FP7-IP project ASCENS on autonomic service-component ensembles. The contact person is Martin Wirsing (LMU München, Germany).
- The FP7-IST coordination action EternalS on trustworthy eternal systems via evolving software, data and knowledge. The action coordinator is Alessandro Moschitti (University of Trento, Italy). The four FP7 FET projects participating in the EternalS action are LivingKnowledge, HATS, Connect, SecureChange.
- The FP7-STREP project ParaPhrase on parallel patterns for adaptive heterogeneous multicore systems. The contact person is Kevin Hammond (University of St. Andrews, UK).
- The FP7-IP project PRO3D on programming for future 3D architectures with many cores. The contact person is Christian Fabre (CEA, France).
- The ESF Cost Action IC0701, a European scientific cooperation on formal verification of object-oriented software. The Chair of the action is Bernhard Beckert (Karlsruhe Institute of Technology, Germany).
- The ESF Cost Action IC0901 “Rich-Model Toolkit”, a European scientific cooperation on an infrastructure for reliable computer systems. The Chair of the action is Viktor Kuncak (EPFL, Switzerland).

The proceedings of the previous editions of FMCO have been published as volumes 2852, 3188, 3657, 4111, 4709, 5382, 5751, 6286, and 6957 of Springer's *Lecture Notes in Computer Science*. We believe that these proceedings provide a unique combination of ideas on software engineering and formal methods which reflect the expanding body of knowledge on modern software systems.

Finally, we thank all authors for the high quality of their contributions, and the reviewers for their help in improving the papers for this volume.

July 2012

Bernhard Beckert
Frank de Boer
Marcello Bonsangue
Ferruccio Damiani

Organization

FMCO 2011 was co-located with the *Second International Conference on Formal Verification of Object-Oriented Software* and was organized by the University of Turin, Italy, in collaboration with the Karlsruhe Institute of Technology, Germany, the Centrum voor Wiskunde en Informatica (CWI), Amsterdam, and the Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands.

Program Organizers

Bernhard Beckert	Karlsruhe Institute of Technology, Germany
Ferruccio Damiani	University of Turin, Italy

Organizing Committee

Bernhard Beckert	Karlsruhe Institute of Technology, Germany
Sara Capecchi	University of Turin, Italy
Ferruccio Damiani	University of Turin, Italy
Vladimir Klebanov	Karlsruhe Institute of Technology, Germany
Luca Padovani	University of Turin, Italy

Sponsoring Institutions

COST Action IC0701
Museo Regionale di Scienze Naturali (MRSN), Turin, Italy
University of Turin

Table of Contents

The ASCENS Project

ASCENS: Engineering Autonomic Service-Component Ensembles 1
Martin Wirsing, Matthias Hölzl, Mirco Tribastone, and Franco Zambonelli

A Language-Based Approach to Autonomic Computing 25
Rocco De Nicola, Gianluigi Ferrari, Michele Loreti, and Rosario Pugliese

A Survey on Basic Connectors and Buffers 49
Roberto Bruni, Hernán Melgratti, and Ugo Montanari

The EternalS Coordination Action

Synthesis-Based Variability Control: Correctness by Construction 69
Anna-Lena Lamprecht, Tiziana Margaria, Ina Schaefer, and Bernhard Steffen

Modeling Application-Level Management of Virtualized Resources in ABS 89
Einar Broch Johnsen, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa

HATS Abstract Behavioral Specification: The Architectural View 109
Reiner Hähnle, Michiel Helvensteijn, Einar Broch Johnsen, Michael Lienhardt, Davide Sangiorgi, Ina Schaefer, and Peter Y.H. Wong

Automatic Service Categorisation through Machine Learning in Emergent Middleware 133
Amel Bennaceur, Valérie Issarny, Richard Johansson, Alessandro Moschitti, Romina Spalazzese, and Daniel Sykes

Towards a Model- and Learning-Based Framework for Security Anomaly Detection.. 150
Matthias Gander, Basel Katt, Michael Felderer, and Ruth Breu

Enhancing Model Driven Security through Pattern Refinement Techniques 169
Basel Katt, Matthias Gander, Ruth Breu, and Michael Felderer

Project Zeppelin: A Modern Web Application Development Framework 184
Leigh Griffin, Peter Elger, and Eamonn de Leastar

The ParaPhrase Project

Managing Adaptivity in Parallel Systems ... 199
Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Carlo Montangero, and Laura Semini

The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems ... 218
Kevin Hammond, Marco Aldinucci, Christopher Brown, Francesco Cesarini, Marco Danelutto, Horacio González-Vélez, Peter Kilpatrick, Rainer Keller, Michael Rossbory, and Gilad Shainer

Paraphrasing: Generating Parallel Programs Using Refactoring ... 237
Christopher Brown, Kevin Hammond, Marco Danelutto, Peter Kilpatrick, Holger Schöner, and Tino Breddin

An Abstract Annotation Model for Skeletons ... 257
Marco Aldinucci, Sonia Campa, Peter Kilpatrick, Fabio Tordini, and Massimo Torquati

The PRO3D Project

PRO3D, Programming for Future 3D Manycore Architectures: Project's Interim Status ... 277
Christian Fabre, Iuliana Bacivarov, Ananda Basu, Martino Ruggiero, David Atienza, Éric Flamand, Jean-Pierre Krimm, Julien Mottin, Lars Schor, Pratyush Kumar, Hoeseok Yang, Devesh B. Chokshi, Lothar Thiele, Saddek Bensalem, Marius Bozga, Luca Benini, Mohamed M. Sabry, Yusuf Leblebici, Giovanni De Micheli, and Diego Melpignano

Thermal-Aware Task Assignment for Real-Time Applications on Multi-Core Systems ... 294
Lars Schor, Hoeseok Yang, Iuliana Bacivarov, and Lothar Thiele

Component Assemblies in the Context of Manycore ... 314
Ananda Basu, Saddek Bensalem, Marius Bozga, Paraskevas Bourgos, Mayur Maheshwari, and Joseph Sifakis

Low-Cost Dynamic Voltage and Frequency Management Based upon Robust Control Techniques under Thermal Constraints ... 334
Sylvain Durand, Suzanne Lesecq, Edith Beigné, Christian Fabre, Lionel Vincent, and Diego Puschini

Author Index ... 355

ASCENS: Engineering Autonomic Service-Component Ensembles

Martin Wirsing[1], Matthias Hölzl[1],
Mirco Tribastone[1], and Franco Zambonelli[2]

[1] Institut für Informatik
Ludwig-Maximilians-Universität München
{martin.wirsing,matthias.hoelzl,mirco.tribastone}@ifi.lmu.de
[2] Department of Science and Engineering Methods
University of Modena and Reggio Emilia
franco.zambonelli@unimore.it

Abstract. Today's developers often face the demanding task of developing software for ensembles: systems with massive numbers of nodes, operating in open and non-deterministic environments with complex interactions, and the need to dynamically adapt to new requirements, technologies or environmental conditions without redeployment and without interruption of the system's functionality. Conventional development approaches and languages do not provide adequate support for the problems posed by this challenge. The goal of the ASCENS project is to develop a coherent, integrated set of methods and tools to build software for ensembles. To this end we research foundational issues that arise during the development of these kinds of systems, and we build mathematical models that address them. Based on these theories we design a family of languages for engineering ensembles, formal methods that can handle the size, complexity and adaptivity required by ensembles, and software-development methods that provide guidance for developers. In this paper we provide an overview of several research areas of ASCENS: the SOTA approach to ensemble engineering and the underlying formal model called GEM, formal notions of adaptation and awareness, the SCEL language, quantitative analysis of ensembles, and finally software-engineering methods for ensembles.

1 Introduction

The increasing miniaturization and decreasing cost of computers and microcontrollers has led to nearly ubiquitous adoption of software-intensive systems. Traditional computer systems such as notebooks, workstations and servers are networked with huge numbers of physical appliances and devices that rely heavily on software, such as smartphones, industrial controllers, and smart robots. We want these systems to integrate seamlessly into our lives and environments, and we want them to responsibly utilize available resources without compromising our privacy or security.

B. Beckert et al. (Eds.): FMCO 2011, LNCS 7542, pp. 1–24, 2013.
© Springer-Verlag Berlin Heidelberg 2013

1.1 What Are Ensembles?

Numerous reasons why it is not only desirable but necessary to develop these kinds of systems have been documented [12]. In this context the ICT-FET project InterLink [14] has coined the term *ensemble* for a particularly interesting class of systems: Ensembles are software-intensive systems with massive numbers of nodes or complex interactions between nodes, operating in open and non-deterministic environments in which they have to interact with humans or other software-intensive systems in elaborate ways. Ensembles have to dynamically adapt to new requirements, technologies or environmental conditions without redeployment and without interruption of the system's functionality, thereby blurring the distinction between design-time and run-time.

National infrastructures such as the power grid, large online businesses such as Amazon or Google, or the systems used by modern armies, all satisfy the definition of an ensemble. However, as complicated and difficult to build as these systems are, they solve relatively well-understood problems and their size is to a large degree a function of the amount of their scale, as well as the data and the number of transactions they have to process. These are interesting and complex problems, but they are far from the largest challenges when building ensembles as defined in the previous systems: None of these systems can actually adapt to unforeseen environmental conditions in a meaningful way; and none of these systems can easily evolve to satisfy new requirements.

1.2 The ASCENS Approach

Instead of static software that operates without knowledge about its environment and hence relies on manual configuration and optimization we have to build systems with self-aware, intelligent components that mimic natural features like adaptation, self-organization, and autonomous as well as collective behavior. However, traditional software engineering, both agile and heavyweight, relies to a large degree on code inspection and testing, approaches which are not adequate for reliably developing large concurrent systems, let alone self-aware, adaptive systems. Formal methods have successfully been employed in an ever increasing number of projects; however, they generally cannot deal with the dynamic and open-ended nature of the systems we are interested in, and they are difficult to scale to the size of industrial-scale projects. Approaches from autonomic and multi-agent systems address aspects such as self-configuration and self-optimization, but they often lack necessary guarantees for reliability, dependability and security and are therefore not easily appropriate for critical systems without modification.

One of the most important and challenging duties when engineering ensembles is to ensure that an ensemble can continue to work reliably in spite of unforeseen changes in its environment and requirements, and that adaptation does not lead to the system becoming inoperable, unsafe or insecure. To achieve this goal, the ASCENS project researches ways of building ensembles that combine the maturity and wide applicability of traditional software-engineering approaches

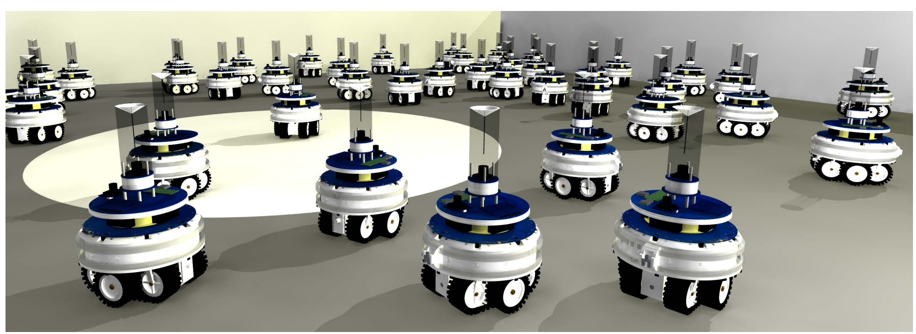


Fig. 1. Ensemble of robots

with the assurance about functional and non-functional properties provided by formal methods and the flexibility, low management overhead, and optimal utilization of resources promised by autonomic, self-aware systems. At the core of this research are new concepts for the design and development of autonomous, self-aware systems with parallel and distributed components. We are developing sound, formal reasoning and verification techniques to support the specification and development of these systems as well as their analysis at run-time. The project goes beyond the current state of the art in solving difficult problems of self-organization, self-awareness, autonomous and collective behavior, and resource optimization in a complex system setting. The relevant disciplines use different formalisms and techniques that have to be related in a single framework in order to present ensemble engineers with a unified development approach.

In this paper we present first steps towards the unified approach of ASCENS. Using a simple swarm robot example (Sect. 2) of autonomous robots we highlight several phases in the development of ensembles. The *State Of The Affairs (SOTA)* method (Sect. 3) is our approach to specifying the overall domain and the requirements of an ensemble. The denotational *General Ensemble Model (GEM)* (Sect. 4) serves as semantic basis of SOTA, refines some of the model assumptions of SOTA and provides the semantic foundations for the formal notions of black-box adaptation and awareness (Sect. 5). The *Service Component Ensemble Language* (*SCEL*, Sect. 6) is developed to provide programming support and formal reasoning of the behavior of autonomic components. As an example, we show how to use continuous-time Markov chains and ordinary differential equations for quantitative reasoning of the robot ensemble (Sect. 7). Finally, we discuss a pattern-based approach for engineering ensembles (Sect. 8).

2 Example: Garbage-Collecting Robots

As a running example we will use a swarm of robots that collects garbage in a rectangular exhibition hall (cf. Fig. 1). The robots should move around the room, pick up the garbage that visitors have dropped and move it to the service

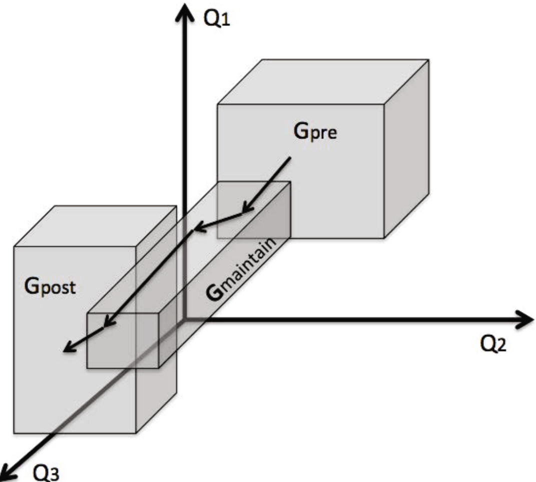


Fig. 2. The trajectory of an entity in the SOTA space, starting from a goal precondition and trying to reach the postcondition while moving in the area specified by the maintain condition

area. For simplicity we assume that the service area is just a rectangular strip along one side of the hall. Since visitors do not want to be distracted by too many robots driving around the exhibition area, there should be as few robots outside the service area as possible while still keeping the hall adequately clean. Furthermore, for environmental and cost reasons the robots should minimize the amount of energy that the swarm consumes. Depending on the sensors of the robots and the type of garbage they are collecting, they may be able to perceive garbage from some distance away, or they may be perceive garbage only while they are driving over it.

3 SOTA: Domain and Requirements Modeling

SOTA (State Of The Affairs) [1] is the ASCENS approach to describing the overall domain and the requirements for a system. In SOTA we identify the behavior of a system with a single trajectory through a *state space*. The state space is the set of all possible states of the system at a single point of time, the trajectory describes how the state varies over time during an actual execution of the system.

Each point in the system's state space corresponds to a *state of the affairs.* In SOTA, the state of the affairs thus represents the state of all parameters that may affect the ensemble's behavior and that are relevant to its capabilities. Although it is common practice to distinguish between the ensemble and the environment (and thus to distinguish between the parameters that describe some characteristics of the environment and those that are inherent of the system),

such a distinction can often blur in complex situated ensembles. Accordingly, in general SOTA does not make such a distinction. In SOTA, we consider the case in which the state space is a finite product $\mathbf{Q} = Q_1 \times \cdots \times Q_n$.

As the system executes, its position in the state space changes either due to the actions of the ensemble or because of the dynamics of the environment. Thus, each execution of a system gives rise to a trajectory ξ in its state space (see Fig. 2).

When modeling requirements we are usually interested in "what the system should do." In the SOTA model, this corresponds to achieving or maintaining certain states of the affairs throughout the systems execution, or more formally to specifying one or more regions of the system's set of all possible trajectories in which the system's observed trajectory has to remain. It is often convenient to specify these regions in the form of *goals*: A goal G in SOTA is a triple of the form $G = \langle G_{pre}, G_{post}, G_{maintain} \rangle$ consisting of a precondition G_{pre} that specifies in which states of the affairs G should become active, a postcondition G_{post} that specifies when the goal has been achieved, and a condition $G_{maintain}$ that has to be maintained while the goal is active. The maintain condition is often called *utility* in SOTA, but in this paper we reserve the term utility for the more general definition given in Sect. 4. A trajectory therefore satisfies a goal G if, whenever the precondition G_{pre} is satisfied, the trajectory stays in the region of the state space specified by $G_{maintain}$ until the postcondition G_{post} is satisfied. After that, the goal has been reached and is no longer relevant for the system execution until its precondition becomes activated again. This capability of pursuing goals during a system execution naturally matches goal-oriented and intentional entities (e.g., humans, organizations, and multi-agent systems), and consequently autonomic and self-adaptive systems.

It is possible to model systems at various levels of detail using the SOTA approach. A very simple model of the robot ensemble that was described in Sect. 2, under the assumption that we have a fixed number N of robots, might be defined as follows: The state space consists of the states of the individual robots, a count $g^\sharp$ of the items of garbage currently in the public part of the exhibition area, and a boolean flag $o^\flat$ that indicates whether the exhibition is currently open for the public or not. We describe each robot by its position in the exhibition area, p_i, and its state s_i. The state can either be $\mathsf{Resting}$, $\mathsf{Searching}$, or $\mathsf{Carrying}$, depending on whether the robot is currently resting, searching for garbage, or carrying a garbage item back to the service area.

Accordingly, the state of the affairs space of the robot ensemble can be described as follows:

$$\begin{aligned}
p_i &= \langle x_i, y_i \rangle \in \mathbb{R} \times \mathbb{R} && \text{Position of robot } i \\
\mathsf{Area} &\subseteq \mathbb{R} \times \mathbb{R} && \text{Exhibition Area} \\
s_i &\in \{\mathsf{Searching}, \mathsf{Resting}, \mathsf{Carrying}\} && \text{State of robot } i \\
g^\sharp &\in \mathbb{N} && \text{Number of garbage items} \\
o^\flat &\in \mathbb{B} && \text{Exhibition open for public?} \\
\mathbf{Q} &= \{\langle p_1, s_1, \ldots, p_N, s_N, g^\sharp, o^\flat \rangle \mid p_i \in \mathsf{Area}\} && \text{State space}
\end{aligned}$$

One of our goals might be to always have fewer than 300 garbage items in the exhibition area while the exhibition is open. This could be described by the following goal G^1:

$$\begin{aligned} G^1_{pre} &\equiv o^\flat = true \\ G^1_{maintain} &\equiv g^\sharp < 300 \\ G^1_{post} &\equiv o^\flat = false \end{aligned}$$

G^1 states that whenever the exhibition opens (i.e., $o^\flat$ becomes *true*), the number of garbage items on the floor, $g^\sharp$, is less than 300. Once the exhibition closes, the postcondition of the goal becomes *true* and the goal is abandoned until the exhibition opens again.

4 GEM: The General Ensemble Model

In SOTA we are concerned with the overall domain and the requirements of the system. For this it is sufficient to deal with the state of the affairs without regard for details such as the state's internal structure or the probabilities of the different trajectories.

For a more detailed investigation of the structure and behavior of ensembles we need a more expressive model. To this end, in parallel with the definition of the SOTA model and in concert with it, we have defined the *General Ensemble Model (GEM)* [13], to model the behavior of ensembles in the state-of-the-affairs space. In Sect. 4.1 we introduce the notion of trajectory space on which the GEM model is based; in Sect. 4.2 we show how goals and utilities are used in GEM. Finally, we give a brief introduction to a probabilistic extension of GEM in Sect. 4.3.

4.1 The Trajectory Space

As in SOTA, in GEM it is not necessary to distinguish between ensemble and environment. However, whenever it is necessary to do that, or when the system specification enforces such a distinction, a unique state space can always be obtained by combining ensemble and environment using a so-called *combination operator*; in the following sections we will use the term *system* to refer to this combination. Combination operators are also used as the means to hierarchically build ensembles from simpler components and smaller ensembles; therefore they serve as a uniform way to model a system's structure and behavior.

In general, a system can behave in a non-deterministic manner and therefore have multiple possible trajectories through the state space. If we know all possible trajectories of the system we know everything that the state space can express about the system. In GEM we identify a system **S** with the set of all its

possible trajectories in the SOTA space. We call the space of all trajectories the *trajectory space* Ξ.[1] Then, a system is a subset of the trajectory space, $\mathbf{S} \subseteq \Xi$.

The state of the affairs concept of SOTA can therefore also be expressed in an enriched way to account for such trajectories: for each trajectory ξ of the system, and at each point in time t the state of affairs is the value $\mathbf{S}(\xi, t)$, which is a point of the state space $\mathbf{Q}$:

$$\mathbf{S}(\xi, t) = \xi(t) = \langle q_i \rangle_{i \in I} \in \mathbf{Q} \quad \text{if } \xi \in \mathbf{S}.$$

In GEM we structure the state space as the result of an interaction between the ensemble and its environment. We formalize this using the notion of combination operator: let Ξ^{ens} and Ξ^{env} be the trajectory spaces of the ensemble and environment, respectively[2], and let $\otimes : \Xi^{ens} \times \Xi^{env} \to \Xi$ be a partial map that is a surjection onto $\mathbf{S}$, i.e., there exist $\mathbf{S}^{ens} \subseteq \Xi^{ens}$ and $\mathbf{S}^{env} \subseteq \Xi^{env}$ such that $\mathbf{S}^{ens} \otimes \mathbf{S}^{env} = \mathbf{S}$. In this case we obtain a trajectory of the system for compatible pairs of ensemble and environment trajectories in $\mathbf{S}^{ens} \times \mathbf{S}^{env}$. We therefore regard the system as the result of combining ensemble $\mathbf{S}^{ens}$ and environment $\mathbf{S}^{env}$ using the operator $\otimes$.

For example, in GEM we can structure a model of the garbage-collecting robot ensemble as follows: we define the state space $\mathbf{Q}^{robot}$ and the trajectory space Ξ^{robot} as

$$\begin{aligned} \mathbf{Q}^{robot} &= \mathbb{R}^2 \times \{\mathsf{Searching}, \mathsf{Resting}, \mathsf{Carrying}\} \\ \Xi^{robot} &= \mathcal{F}[T \to \mathbf{Q}^{robot}]. \end{aligned}$$

The model of each robot $\mathbf{S}_i^{robot}$ is a subset of the trajectory space consisting of all possible trajectories that the robot can take through its state space:

$$\mathbf{S}_i^{robot} \in \mathfrak{P}(\Xi^{robot}).$$

The ensemble consisting of all N robots has as state space $\mathbf{Q}^{ens} = (\mathbf{Q}^{robot})^N$, and as trajectory space

$$\Xi^{ens} = \mathcal{F}[T \to \mathbf{Q}^{ens}],$$

and the model of the ensemble, $\mathbf{S}^{ens}$, can be obtained from the models of the individual robots by a combination operator

$$\otimes : \mathfrak{P}(\Xi^{robot})^N \to \mathfrak{P}(\Xi^{ens})$$

that combines all trajectories of robots that are physically possible, i.e., $\otimes$ is essentially the canonical map between $\mathfrak{P}(\Xi^{robot})^N$ and $\mathfrak{P}(\Xi^{ens})$, but it removes those trajectories where robots would overlap in space.

[1] For the mathematically inclined reader, we point out that $\Xi = \mathcal{F}[T \to \mathbf{Q}]$, where T is the time domain and $\mathcal{F}[T \to \mathbf{Q}]$ the set of all functions from T to $\mathbf{Q}$.

[2] Formally we have $\Xi^{ens} = \mathcal{F}[T \to Q^{ens}]$ where $Q^{ens} = \prod_{k \in K} Q_k^{ens}$, and $\Xi^{env} = \mathcal{F}[T \to Q^{env}]$ where $Q^{env} = \prod_{l \in L} Q_l^{env}$. Note that the sets Q_k^{ens} and Q_l^{env} may be different from the sets Q_i that appear in the system's state space $\mathbf{Q} = \prod_{i \in I} Q_i$.

In this example, we define a more detailed state space for the environment than we did in the previous section. We again include a boolean value $o^\flat$ indicating whether the exhibition is open for the public, and the number of garbage items in the area $g^\sharp$. In addition we add a function $g : \mathbb{N} \to \mathbb{R}^2$ so that $g(i)$ gives the location of the i-th garbage item for $1 \leq i \leq g^\sharp$, and the coordinates of the public exhibition area and the service area:

$$\mathbf{Q}^{env} = \mathbb{B} \times \mathbb{N} \times \mathcal{F}[\mathbb{N} \to \mathbb{R}^2] \times \mathfrak{P}(\mathbb{R}^2) \times \mathfrak{P}(\mathbb{R}^2).$$

As usual, the trajectory space of the environment is $\varXi^{env} = \mathcal{F}[T \to \mathbf{Q}^{env}]$ and each environment $\mathbf{S}^{env}$ is a member of $\mathfrak{P}(\varXi^{env})$. In this simple example, the state space $\mathbf{Q}$ for the whole ensemble is the product $\mathbf{Q}^{ens} \times \mathbf{Q}^{env}$ and the ensemble's trajectory space is defined as $\mathcal{F}[T \to \mathbf{Q}]$; the combination operator for ensemble and environment has then the signature

$$\otimes : \mathfrak{P}(\varXi^{ens}) \times \mathfrak{P}(\varXi^{env}) \to \mathfrak{P}(\varXi)$$

and combines again all trajectories of environment and ensemble that are possible while removing those combined trajectories that cannot happen (e.g., no robot can be outside the exhibition area, a robot's state can only change from Searching to Carrying when it is over a garbage item, and if no robot is in state Searching during a time interval $[t_0, t_1]$, then the number of garbage items cannot decrease between t_0 and t_1, etc.).

4.2 Goals and Utilities

GEM is intended to serve as a semantic foundation for various kinds of calculi and formal methods which often have a particular associated logic. We define the notion of goal satisfaction "System $\mathbf{S}$ satisfies goal G," written $\mathbf{S} \models G$ in a manner that is parametric in the logic and in such a way that different kinds of logic can be used to describe various properties of a system. See [13] for details.

While goals allow us to express many requirements of systems, many authors have observed that "[g]oals alone are not enough to generate high-quality behavior in most environments." [20]. For example, the property "the garbage-collecting robots should use as little energy as possible" cannot be expressed as a goal, since there is no hard boundary on energy consumption that tells us whether the goal was achieved or not. Instead we have to compare the energy consumption along various trajectories and rate trajectories with lower consumption as better than ones with higher consumption. A trajectory ξ of the system may therefore be more or less desirable; we assign a measure $u(\xi)$ to each trajectory so that $u(\xi_i) \preceq u(\xi_j)$ if and only if ξ_j is at least as desirable as ξ_i. The function u is called the *utility function*, and $u(\xi)$ is called the *utility* of trajectory ξ. Often, the definition of utilities is complicated by having not just a single criterion that we want to optimize, but rather various conflicting criteria between which we have to achieve a trade-off. In our example, the requirement to achieve the "best" compromise between the number of robots in the hall and

the amount of garbage cleaned up is an instance of such a multi-criteria decision problem [15]. Solutions for these kinds of trade-off can be achieved using the framework of utilities as well; see Sect. 7.3 for a more detailed discussion.

An optimization goal is then a goal that requires the optimization of a utility. This may take the form of either optimizing the maximal achieved utility at some point on a trajectory through the state space, or the goal may be to optimize an aggregate utility along the trajectory.

Note that utilities are strictly more expressive than goals; in fact it is often useful to interpret goals as utilities as well: We can transform each goal G into a utility u_G with the value 1 for each trajectory ξ that satisfies the goal and the value 0 for all other trajectories. Then, optimizing this goal has the same effect as satisfying the original goal; only trajectories that satisfy the goal are taken if such trajectories exist. However, utilities are more flexible than goals: If, for example, G is the goal that no robot should run out of energy, we can define u_G to assign values between 0 and 1 to trajectories that sometimes violate G, depending on the average number of robots that run out of energy every day. Then, even if G cannot be permanently satisfied, the ensemble can choose the trajectory that violates the goal for the least amount of time.

4.3 Probabilistic GEM

The model presented so far is sufficient to deal with deterministic and non-deterministic systems. However, for many practical purposes, simply knowing the possible trajectories of a system is not enough; instead, we need to know the probability for taking particular trajectories to evaluate the quality of the system. Therefore we need to turn to stochastic models. This would be needed, for example, to capture the situation where a robot receives sensor input with measurement errors.

Thus we assume that a probability measure $\mathbf{P}(\mathbf{X})$ is given for each set of trajectories $\mathbf{X}$.[3] $\mathbf{P}(\mathbf{X})$ describes the probability that a trajectory in $\mathbf{X}$ is taken by the system. If the system is generated from an ensemble $\mathbf{S}^{ens}$ and an environment $\mathbf{S}^{env}$, then we assume that probability distributions over their respective trajectory spaces are given, and that the combination operator $\otimes$ computes the distribution of $\mathbf{S}$ from these.

Given a probability measure $\mathbf{P}$ and a utility function u for a system $\mathbf{S}$, we define the *evaluation* of a system $\mathbf{S}$ as the expected utility, i.e.,

$$eval_u(\mathbf{S}) = E_{\mathbf{P}}[\mathbf{S}, u] = \int_{\xi \in \mathbf{S}} \mathbf{p}(\xi) u(\xi) d\xi.$$

where $\mathbf{p}$ is the probability density of $\mathbf{P}$. The evaluation gives us an easy criterion to compare different systems: a system $\mathbf{S}_1$ has a better utility than a system $\mathbf{S}_2$ if its evaluation is higher.

[3] More precisely, we assume that a probability space is given, i.e., that we have a σ algebra Σ over Ξ and a probability measure on Σ. In this overview paper we will ignore these kinds of technical complications.

In the next section we will define adaptation and awareness based on the notions developed in this section.

5 Adaptation and Awareness

Using the GEM model for ensembles presented in the previous section, we can define mathematical models for the important notions of adaptation (Sect. 5.1 and awareness (Sect. 5.2).

5.1 Adaptation

There are various senses in which the word "adaptation" is used, but an important characteristic of adaptation is the ability to react usefully to some kind of change. We can describe this reaction either by looking "inside" the system in order to describe the mechanism by which the system implements the changes it performs, or we can look at the system by evaluating only the quality of the system's behavior, without describing the mechanisms by which it is achieved. We call the first approach white-box or glass-box adaptation; it is further described in [3], in the following we focus on the second approach which we call *black-box adaptation*.

We call a set of environments $\mathbf{A}^{env}$ together with a goal G (and possibly a probability measure) an *adaptation domain* $\mathbf{A}$. The adaptation domain represents the situations in which we want the ensemble to work. Furthermore, we suppose that we can define a combination operator $\otimes$ that combines any environment $\mathbf{S}^{env} \in \mathbf{A}^{env}$ with an ensemble $\mathbf{S}^{ens}$. We then say that $\mathbf{S}^{ens}$ *can adapt to* $\boldsymbol{A}$, written $\mathbf{S}^{ens} \Vdash \mathbf{A}$:

$$\mathbf{S}^{ens} \Vdash \mathbf{A} \iff \forall \mathbf{S}^{env} \in \mathbf{A}^{env} : \mathbf{S}^{ens} \otimes \mathbf{S}^{env} \models G.$$

In the case of probabilistic systems we replace the goal G with a utility u in the definition of adaptation domains and lift the evaluation function *eval* to an adaptation domain, so that instead of the evaluation of the system, $eval_u(\mathbf{S})$, we obtain the *evaluation with respect to an adaptation domain* or *lifted evaluation* $\mathbf{eval}(\mathbf{S}^{ens}, \mathbf{A})$. In the simplest case, **eval** might be the minimal or maximal evaluation of $\mathbf{S}^{ens} \otimes \mathbf{S}^{env}$ for all $\mathbf{S}^{env} \in \mathbf{A}^{env}$. It is often useful to equip the adaptation domain with a probability distribution and define the lifted evaluation as the expected value of the evaluation for all environments in $\mathbf{A}^{env}$.

The environment models in the previous section allow us to define a wide range of adaptation domains. For example, we could have adaptation domains that vary parameters of the environment, such as the size or topology of the exhibition area, or the distribution of the garbage. We can also define adaptation domains with different goals, e.g., the maximum number of garbage items that are allowed. Let $\mathbf{A}_l^{env}$ be the set of all square arenas with side length l containing no obstacles and in which garbage items appear according to some distribution. Let goal $G^{<n}$ be the property that fewer than n garbage items are in the arena while the exhibition is open (the example in Sect. 3 corresponds to $n = 300$).

We can then define adaptation domains $\mathbf{A}_l^{<n} = \langle \mathbf{A}_l^{env}, G^{<n} \rangle$. We define the combination operator $\otimes$ such that it causes a robot to pick up a dropped garbage item whenever the robot passes over the garbage item while being in state Searching. The relation $\mathbf{S}^{ens} \Vdash \mathbf{A}_l^{<n}$ then holds for each Ensemble $\mathbf{S}^{ens}$ if and only if every trajectory of the ensemble in a square arena with side length l leaves fewer than n garbage items in the arena while the exhibition is open. A further refinement would be to consider an adaptation domain that uses a utility function to rank ensembles according to their energy consumption. For a practical example, see Sec. 7.3.

Adaptation domains allow us to compare the ability of different ensembles to adapt to a given range of situations. To simplify this comparison, we consider sets of adaptation domains which we call *adaptation spaces*. Given an adaptation space $\mathcal{A}$ we can compare the ability of ensembles to adapt by set-theoretic inclusion:

$$\mathbf{S}_2^{ens} \sqsubseteq \mathbf{S}_1^{ens} \iff \forall \mathbf{A} \in \mathcal{A} : \mathbf{S}_2^{ens} \Vdash \mathbf{A} \implies \mathbf{S}_1^{ens} \Vdash \mathbf{A}$$

or in the case of utilities

$$\mathbf{S}_2^{ens} \sqsubseteq \mathbf{S}_1^{ens} \iff \forall \mathbf{A} \in \mathcal{A} : \mathbf{eval}(\mathbf{S}_2^{ens}, \mathbf{A}) < \mathbf{eval}(\mathbf{S}_1^{ens}, \mathbf{A})$$

In this case, we say that $\mathbf{S}_1^{ens}$ is *at least as adaptive* as ensemble $\mathbf{S}_2^{ens}$ for $\mathcal{A}$; if we additionally have $\mathbf{S}_1^{ens} \not\sqsubseteq \mathbf{S}_2^{ens}$ we say that $\mathbf{S}_1^{ens}$ is *more adaptive* than $\mathbf{S}_2^{ens}$.

For example, we can define the adaptation spaces $\mathcal{A}_l = \{\mathbf{A}_l^{<n} \mid n \in \mathbb{N}\}$ which ranks the adaptivity of ensembles according to their ability to collect garbage in an arena of side length l, $\mathcal{A}^{<n} = \{\mathbf{A}_l^{<n} \mid l \in \mathbb{R}\}$ which ranks ensembles by their ability to achieve a certain level of cleanliness in arenas of varying sizes and $\mathcal{A} = \{\mathbf{A}_l^{<n} \mid n \in \mathbb{N}, l \in \mathbb{R}\}$ which combines these two criteria.

The previous notion of adaptation assumes that the goal that we want the ensemble to achieve is fixed for all environments. This may lead to very complicated goal specifications if we want to consider, e.g., quality-of-service properties that depend on the environment. To this end, we extend the notion of adaptation space and define a *generalized adaptation domain* as the set consisting of pairs of environments and goals, $\mathbf{A} = \{\langle \mathbf{S}_i^{env}, G_i \rangle \mid i \in I\}$. Adaptation to a generalized adaptation domain then means

$$\mathbf{S}^{ens} \Vdash \mathbf{A} \iff \forall \langle \mathbf{S}_i^{env}, G_i \rangle \in \mathbf{A} : \mathbf{S}^{ens} \otimes \mathbf{S}_i^{env} \models G_i.$$

The notions of adaptation space and lifted evaluation can be extended to *generalized adaptation spaces* in the obvious manner.

5.2 Awareness

One of the most important notions for adaptive systems is "awareness." Intuitively, this term denotes an internal representation that the ensemble has about some aspect of itself or its environment which is kept up-to-date as the system

moves through the state space. This does not necessarily imply that the ensemble immediately registers changes in this aspect, it is also sufficient if, e.g., the system receives periodic updates about changes from sensors or other systems.

In contrast to adaptation, it is our opinion that a definition of awareness has to refer to the internal representation of the ensemble, and, if possible, it should also take into account the information about the environment that the ensemble derives from its internal representation. For example, if a garbage-collecting robot has an exact internal representation of all the pieces of garbage in the arena, but no internal interpretation of this representation, it seems to us that it is not justified to call that robot "aware of the locations of pieces of garbage." How to determine whether the robot has an interpreted internal representation is obviously a difficult problem. In this paper we restrict ourselves to the simplest (but highly unrealistic) case in which we have a function giving, for each state of its internal awareness representation, the states of the affairs that the system considers possible.[4] In more realistic scenarios we can sometimes estimate this set of possible states of the affairs from our knowledge of the ensemble's implementation or by observing the ensemble's behavior.

More formally, let $Q^{ens} = \prod_{k \in K} Q_k^{ens}$ be the state space of the ensemble, $\Xi^{ens} = \mathcal{F}[T \to Q^{ens}]$ its trajectory space, and $J \subseteq K$ such that the Q_j, $j \in J$ are the components of Q^{ens} relevant for the awareness of the ensemble. We then call $\mathbf{B} = \mathcal{F}[T \to \prod_{j \in J} Q_j]$ the *awareness section* of Ξ^{ens}. We write $\xi|_{\mathbf{B}}$ for the obvious restriction of a trajectory in Ξ^{ens} to $\mathbf{B}$. As mentioned in the previous paragraph, we assume that we have a function $\varepsilon_{\mathbf{S}} : \mathbf{B} \times T \to \mathfrak{P}(\Xi)$, which we call the *awareness function*, that gives the trajectories that the ensemble considers possible for each value of its awareness section at each point in time.

With this definition, we can say what it means for the awareness function to be correct: let $\xi^{ens} \in \Xi^{ens}$, $\xi^{env} \in \Xi^{env}$ such that $\xi = \xi^{ens} \otimes \xi^{env}$ exists. If for every time $t \in T$ the actual state of the affairs at time t, $\xi(t)$, is in the set of possible states of the affairs according to the system's awareness function $\varepsilon_{\mathbf{S}}$, i.e., $\xi(t) \in \varepsilon_{\mathbf{S}}(\xi^{ens}|_{\mathbf{B}}, t)$ then $\varepsilon_{\mathbf{S}}$ is *correct for* $\langle \xi^{ens}, \xi^{env} \rangle$. If the awareness is correct for all pairs $\langle \xi^{ens}, \xi^{env} \rangle$ for which $\xi^{ens} \otimes \xi^{env}$ is defined, then it is *globally correct.*

In the GEM definition of the robot example given in Sect. 4, the awareness section $\mathbf{B}_{robot}$ for each robot might, e.g., consist of its position and internal state (Searching, Resting or Carrying) and the number of garbage items in the arena. If the awareness function ε_{robot} is the function mapping, for each time t, the trajectory $\xi|_{\mathbf{B}_{robot}}$ into the set of all trajectories of the ensemble which agree with the argument on $\mathbf{B}_{robot}$, then the awareness function of the robot is correct. In this case the robot is precisely aware of its own state, but not of the state of the environment, even though the exact number of garbage items is contained in its awareness section.

[4] Giving the set of possible states of the affairs is in practice not particularly useful. It is much more practical to give a probability distribution over the set of possible states of the affairs. However, since this change introduces significant mathematical complexities, we restrict the presentation to the deterministic case in this overview.

To compare different ensembles operating in the same environment it is also useful to define some additional notions: Let $\xi^{env} \in \Xi^{env}$ and let $\mathbf{S}^{ens}[\xi^{env}]$ be the set of all trajectories $\xi^{ens} \in \mathbf{S}^{ens}$ such that $\xi^{ens} \otimes \xi^{env}$ exists. If, for all $\xi^{ens} \in \mathbf{S}^{ens}[\xi^{env}]$, the awareness function $\varepsilon_{\mathbf{S}}$ is correct for $\langle \xi^{ens}, \xi^{env} \rangle$, then we say that it is *correct with respect to environment trajectory* ξ^{env}. We define the *environmental awareness function* $\varepsilon_{\mathbf{S}}^{env}$ as

$$\varepsilon_{\mathbf{S}}^{env} : \xi^{env} \mapsto \bigcup_{\xi^{ens} \in \mathbf{S}^{ens}[\xi^{env}]} \varepsilon_{\mathbf{S}}(\xi^{ens}|_{\mathbf{B}}, t) \,|_{\Xi^{env}} .$$

The environmental awareness function $\varepsilon_{\mathbf{S}}^{env}$ can be used to compare the awareness of different ensembles operating in the same environment. The definitions of correctness transfer *mutatis mutandis* to environmental awareness; this notion of correctness only judges whether the ensemble is aware of the environment and not of its internal state. For two ensembles $\mathbf{S}_1$ and $\mathbf{S}_2$ and an environment trajectory ξ^{env}, we say that the awareness of $\mathbf{S}_1$ with respect to ξ^{env} is *more precise* than that of $\mathbf{S}_2$ if $\varepsilon_{\mathbf{S}_1}^{env}(\xi^{env})$ is correct and if $\varepsilon_{\mathbf{S}_1}^{env}(\xi^{env}) \subseteq \varepsilon_{\mathbf{S}_2}^{env}(\xi^{env})$. We say that the environmental awareness of $\mathbf{S}_1$ is *more precise* than that of $\mathbf{S}_2$ if it is more precise with respect to all trajectories in the environment.

It is easy to see that the above notions can be extended to general adaptation spaces and general adaptation domains in a straightforward manner. It is then possible to define a *minimal level of awareness* that an ensemble $\mathbf{S}^{ens}$ has to possess in order to adapt to a general adaptation domain $\mathbf{A}$ based on the ensemble's environmental awareness: if there are pairs $\langle \mathbf{S}_1^{env}, G_1 \rangle \in \mathbf{A}$ and $\langle \mathbf{S}_2^{env}, G_2 \rangle \in \mathbf{A}$ such that G_1 and G_2 cannot be simultaneously satisfied, then for all trajectories ξ^{env} in $\mathbf{S}_1^{env}$, the environmental awareness of $\mathbf{S}^{ens}$ may not include any trajectory in $\mathbf{S}_2^{env}$.

The definition of environmental awareness can be refined in the sense that (i) not the whole environment has to be taken into account for the comparison and (ii) parts of the ensemble's state space may be taken into account for the purposes of the comparison. This can be used to introduce "levels of awareness" in various dimensions, so that we can, e.g., have awareness of single components, of parts of the ensemble, and of the whole ensemble, or as another dimension, awareness of purely spatial relationship versus awareness of social structure.

6 Solution Models

In Sect. 6.1 we provide a brief introduction to SCEL. It shall be used to give an operational semantics to our case study, in Sect. 6.2.

6.1 Introduction to SCEL

The *Service Component Ensemble Language (SCEL)* [5] is developed within ASCENS to provide programming support and formal qualitative and quantitative

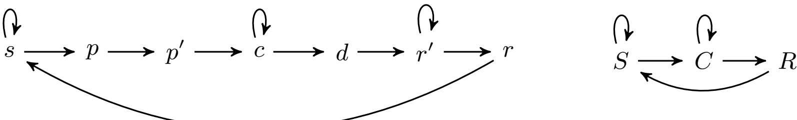
reasoning of the behavior of autonomic components. Its kernel is based on process-algebraic principles, with the usual operators such as action prefix $a.P$, choice $P_1 + P_2$, and recursion via constants $A \triangleq P$. (The syntax for behavioral description is in fact richer, but here we limit ourselves to the fragment which is necessary for the understanding of the remainder.)

A peculiar feature of SCEL is the modeling of actions, which are interactions between a process and a *knowledge repository* where items of information may be accessed. Three types of actions are defined, $\mathbf{get}(T)@c$, $\mathbf{qry}(T)@c$, and $\mathbf{put}(t)@c$, in order to model removal or peek of a *template* T, and insertion of an element t, respectively, into the knowledge repository at the component identified by c. The syntax for templates and tuples are purposely left unspecified and are intended to be specialized by specific instantiations of SCEL; for instance, in the following we consider a Klaim-based approach with tuple spaces [4], although other notions of behavior (e.g., constraint stores) may be similarly devised.

Components are identified by names through *interfaces* with attributes, which have syntax $\mathcal{I}[\mathcal{K}, \Pi, P]$. The attribute $\mathcal{I}.id$ gives the name of the component. This may be used to access its *knowledge manager* $\mathcal{K}$, which handles the knowledge repository (as such it is also left unspecified in SCEL). Policies, defined by Π, are the mechanism to govern the interaction between components—for instance they may be used to regulate access to knowledge repositories (but they are not used later). It is worth of attention to underline that SCEL does not provide a linguistic primitive for the specification of an ensemble; instead, ensembles are inferred from the attributes of the interfaces of components. For instance, in the specification of a component's interface, $\mathcal{I}.ensemble$ is a predicate on interfaces to determine the elements of the ensemble coordinated by the component. Similarly, $\mathcal{I}.membership$ determines the ensembles which the component may join. This design choice allows for a more flexible and dynamic specification than syntactic constructs at the process algebra level.

6.2 SCEL Model of the Case Study

We consider the case study discussed in Section 2 and assume that each garbage-collecting robot behaves independently from the swarm. It explores the exhibition hall in search for items by proceeding along a random direction at a constant velocity. Whilst exploring, it may encounter three kinds of obstacles: another robot or a wall, in which case a collision-avoidance algorithm is invoked to change its direction of movement; or an item, in which case the robot picks it up to return to the service area. This is realized by means of a light source at the service area which is sensed by the robot in order to decide the direction along which to move. When the robot arrives at the service area, it drops off the item and subsequently tries to rest to reduce power consumption. In order to do so, it moves in the service area to find available space, and then goes into sleep mode for some time. When it resumes, it starts exploring the exhibition hall again.



(a) Transition system from the SCEL model (1). (b) Reduced model.

Fig. 3. Qualitative discrete-state behavior of a garbage-collecting robot

Qualitatively, the behavior of a single robot could be modeled with the following SCEL fragment.

$$\begin{aligned}
s &\triangleq \mathbf{get}(\mathit{collision})@\mathit{ctl}.s + \mathbf{get}(\mathit{item})@\mathit{ctl}.p \\
p &\triangleq \mathbf{get}(\mathit{items}, !x)@\mathit{master}.p' \\
p' &\triangleq \mathbf{put}(\mathit{items}, x+1)@\mathit{master}.c \\
c &\triangleq \mathbf{get}(\mathit{collision})@\mathit{ctl}.c + \mathbf{get}(\mathit{arrived})@\mathit{ctl}.d \\
d &\triangleq \mathbf{put}(\mathit{dropped})@\mathit{master}.r' \\
r' &\triangleq \mathbf{get}(\mathit{collision})@\mathit{ctl}.r' + \mathbf{put}(\mathit{sleep})@\mathit{timer}.r \\
r &\triangleq \mathbf{get}(\mathit{elapsed})@\mathit{timer}.s
\end{aligned} \tag{1}$$

The process constants stand for: s = searching for a garbage item; p = picking up item; c = carrying the item (returning to nest); d = dropping off item; r' = searching for a rest place; r = resting. Processes s, c, and r' exhibit similar behavior in that they may consume a tuple *collision* which is produced by some controller *ctl*. However, this results in a self-loop which does not change the behavior as the process behaves as before. Notice that the item is removed from the tuple space, therefore it is responsibility of *ctl* to produce another tuple, whenever a collision is detected. The controller may also produce an *item*, in which case process s behaves then as p. Here, we assume a central repository *master* which keeps track of the total number of items collected during the evolution of the system. The current value is first retrieved and then put back into the repository after being incremented. Another noteworthy process is r', which puts an item *sleep* into the tuple space of a *timer*. It will be able to resume when the timer puts an *elapsed* tuple in its tuple space. The labeled transition system for this process, derived according to the operational semantics of SCEL, is shown in Fig. 3(a). For reasons of space, the (obvious) transition labels are not explicitly given.

The names *ctl*, *master*, and *timer* are assumed to be exposed by other components, k, m, t, respectively (not shown here for brevity), according to the parallel composition

$$(\mathcal{I}_1[\cdot, \Pi, s] \parallel \mathcal{I}_2[\cdot, \Pi, k] \parallel \mathcal{I}_3[\cdot, \Pi, t]) \parallel \mathcal{J}[\cdot, \Pi, m].$$

Here, we are assuming that the tuple spaces at each component are initially empty, and we let Π be the most *permissive* policy which permits access to every tuple space. The definitions of the interfaces are such that $\mathcal{I}_1.id = robot$, $\mathcal{I}_2.id = ctl$, $\mathcal{I}_3.id = timer$, and $\mathcal{J}.id = master$. The other two attributes of an interface, i.e. *ensemble* and *membership*, are taken to be such that all components belong to the same ensemble. This model, which deals with only one robot, can be extended to an arbitrary number of robots by suitably repeating the term between parentheses; the component with interface $\mathcal{J}$ is unique if one assumes a single master node.

For the purposes of quantitative evaluation, the behavior may be simplified by making the following assumptions: (i) The transitions $p \rightarrow p'$ and $p' \rightarrow c$ take up a negligible amount of time with respect to the representative time scales of the system; similarly, (ii) the durations of $d \rightarrow r'$ and $r' \rightarrow r$ are assumed to be negligible. In other words, (i) and (ii) imply that the robot goes to sleep as soon as it enters the service area. The validity of such assumptions was successfully validated with the simulation experiments which are described in the remainder. Overall, these simplifications lead to a smaller discrete-state description as shown in Fig. 3(b). Notice that the three states of this reduced labelled transition system correspond to the states of the robot described in SOTA in Sect. 3, and in GEM in Sect. 4.

7 Quantitative Analysis

In this section, we equip the reduced labelled transition system that arises from the SCEL model with quantitative information, leading to a continuous-time Markov chain, and a compact approximation thereof based on ordinary differential equations, as presented in Sect. 7.1. In Sect. 7.2, we successfully validate the model against simulation. Finally, Section 7.3 uses the model to perform black-box adaption by means of sensitivity analysis over system parameters.

7.1 Quantitative Model

The quantitative model is given in terms of a continuous-time Markov chain (CTMC) that keeps track of the population of robots in each of the states S, C, R, and of the total amount of garbage items to be collected in the exhibition hall, denoted by G. Although the model is defined directly in such an aggregated manner, it can be shown to be automatically inferred from the individual description of a single robot, see Fig. 3(b); this is not discussed here for space reasons (similar arguments to [11] may be used). Thus, each state of the CTMC is associated with a vector of nonnegative integers (S, C, R, G). The chain has the following transitions:

$$(S, C, R, G) \longrightarrow (S-1, C+1, R, G-1), \quad \text{with rate} \quad \mu S \frac{G}{S+C+G}, \tag{2}$$

$$(S, C, R, G) \longrightarrow (S+1, C, R-1, G), \quad \text{with rate} \quad \beta R, \tag{3}$$

$$(S, C, R, G) \longrightarrow (S, C-1, R+1, G), \quad \text{with rate} \quad \gamma C, \tag{4}$$

$$(S, C, R, G) \longrightarrow (S, C, R, G+1), \quad \text{with rate} \quad \lambda. \tag{5}$$

The first transition describes that an item is found; thus, the number of exploring robots is reduced by one and, correspondingly, the number of robots returning to the service area is increased by one; also, the number of items in the exhibition hall decreases. The rate is defined in terms of μ, which is to be intended as the *encounter rate* of each robot, i.e., the opposite of the average time between collisions between robots or between a robot and an item. The actual value used in the model is parametrized by simulation runs. The fraction $G/(S+C+G)$ represents the probability of a successful encounter, which is simply given as the ratio of items with respect to the total amount of objects a robot may encounter. The factor S in the rate is the multiplicative factor in order to consider the rate for the whole population of exploring robots. The robot is assumed to sleep for an exponentially distributed amount of time with rate β, therefore βR is the sleep rate of the overall system. Rate γ is the rate to return to the service area, which is also parametrized with the measurements from simulation. Finally, the last transition denotes drops of garbage items with exponentially distributed inter-arrival times, that is, according to a Poisson process with rate λ.

Although this model may readily be used for the analysis, we observe that it gives rise to an infinite-state Markov chain, even if the total number of robots in each state is always equal to N (those in the initial state of the system), because of (5) which may always increase the number of items. Although this problem can be tackled by numerically truncating the chain, the total number of states grows quickly with N. Using standard manipulations of Eqs. 2–5, it is possible to derive the following system of coupled ordinary differential equations (ODEs) which are interpreted as the first-order approximation of the Markov process.

$$\begin{aligned}
\dot{S} &= -\mu S G (S+C+G)^{-1} + \beta R \\
\dot{C} &= +\mu S G (S+C+G)^{-1} - \gamma C \\
\dot{R} &= +\gamma C - \beta R \\
\dot{G} &= +\lambda - \mu S G (S+C+G)^{-1}
\end{aligned} \tag{6}$$

Together $S(0) = N$, $C(0) = R(0) = G(0) = 0$, this leads to an initial value problem which is easily solved using standard numerical integration. In the following, we consider a scenario with $N = 20$ garbage-collecting robots.

7.2 Validation

A discrete-event simulation of the system under study was developed with the ARGoS tool [19]. The source code for a robot controller was instrumented to

Table 1. Model validation. Steady-state ODE estimates of robot sub-populations against discrete-event simulation of the system.

	S	C	R
Simulation	15.972	3.778	0.250
Model	16.070	3.730	0.200
Normalized error	0.49%	0.24%	0.25%

record the timestamps of transitions according to the classification of states in Fig. 3(b). These logs were used to estimate μ and γ in the model. The former was simply estimated by computing the reciprocal of the average time between two successive timestamps where an encounter with a garbage item or with another robot were registered. In the case of an encounter with the robot, this information was deduced by observing a change of direction in the robot movement, which is the result of the collision-avoidance algorithm. The estimation of γ was performed similarly, by measuring the average time between a robot picking up a garbage item and dropping it off at the service area. With an arena size of 16 squared meters, these parameters were found to be $\mu = 0.012$ and $\gamma = 0.003$. The simulation also logged the total number of robots in each of the states S, C, R as a function of time. Across all experiments, we set $\lambda = 0.010$ and $\beta = 0.050$. The mean steady-state estimates were calculated by using 150 independent runs of the simulation, each lasting ten hours of simulated time. They were compared against the fixed point of the ODE solution. We used the following measure of accuracy to assess the quality of the results:

$$\text{Normalized error} = \frac{|\text{Simulation estimate} - \text{ODE estimate}|}{N} \times 100$$

This error relates the absolute difference with respect to the total population of elements considered in the analysis. This is to better capture the fact that a large absolute difference between two estimates may be practically unimportant when related to the proportions of robots in a particular state. The results of the analysis, shown in Table 1, demonstrate a very good accuracy of the model, with a maximum error less than 0.5% relative to the total population of robots.

7.3 Black-Box Adaptation by Sensitivity Analysis

We now turn to relating this operational interpretation of an ensemble of robots with the SOTA/GEM description. We observe that the trajectory space of GEM simply reduces to the solution of the initial value problem (6), which is unique in this specific model. Furthermore, the general notion of *utility* has here the interpretation of a real-valued function of the solution, i.e. $\varphi(S(t), C(t), R(t), G(t))$. For instance, an interesting utility function is *throughput*, i.e., the frequency at which garbage items are returned to the service area. In this case, as $C(t)$ is the number of returning robots at time t that have picked up a garbage item, throughput may be expressed as the function $\gamma C(t)$.

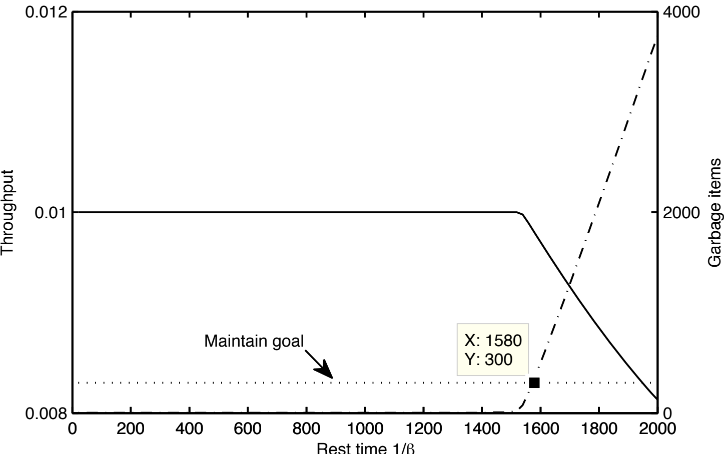


Fig. 4. Sensitivity analysis of throughput of garbage collection (solid line) and dirtiness of exhibition hall (dash-dotted line) against rest time at the service area. The dotted line shows the SOTA requirement (*maintain goal*) that there must not be more than 300 garbage items in the arena.

This dynamic model allows for forms of black-box adaptation as described in a more general sense in Section 5.1. Here, black-box adaptation may realized by means, for instance, of *sensitivity* analysis, intended as the evaluation of different model instances where some parameters of interest are changed in order to study their dependence on the overall system's behavior. For instance, an interesting application would be to evaluate the impact of the resting time on the throughput of garbage collection. Intuitively, the smaller the resting time the higher the throughput, because there will be on average more robots circulating in the arena, all the other parameters remaining the same. However, too high a rate may not be convenient because a robot could be keep exploring the arena without finding garbage items, which are being picked up by all the other robots. This would lead to wasteful consumption of energy, which can be reasonably modeled as a cost function which is linear with the amount of time that a robot is moving.

Therefore, a trade-off is sought between maintaining a clean arena and reducing energy consumption by using robots parsimoniously. One could think of *adapting* the parameters of the robot to ensure that a certain goal of arena cleanliness is maintained. For instance, the SOTA requirement in Section 3 of having less than 300 garbage items in the arena can be translated into a sensitivity analysis which looks at the estimated value of $G(t)$, solution of the system of ODEs. For example, this can be done by inspecting a curve which plots the

steady-state throughput against the average rest time $1/\beta$, as shown in Fig. 4. The throughput curve, in solid line, shows insensitivity for a wide range of rest times, until about 1500. This is because, in those situations, the arena is kept relatively clean (with about 2 garbage items, dash-dotted line), therefore many robots keep exploring but they encounter an item infrequently. As rest times are further increased, however, the robots cannot keep up with the waste; throughput decreases because fewer robots are present, and the arena becomes more soiled. Thus, the maximum allowed rest time predicted by the model, corresponding to the lowest energy consumption possible whilst achieving the desired *maintain goal*, corresponds to 1580, when the garbage-item line and the maintain-goal one (dotted line) intersect.

8 Engineering Ensembles

The previous sections presented techniques and formal foundations for designing and analyzing ensembles. To be useful to the developer, they have to be integrated into the development process. To facilitate this, we propose patterns and best practices for applying our methods in Sect. 8.1, an approach to awareness- and knowledge-cognizant software engineering in Sect. 8.2, and tool support in Sect. 8.3.

8.1 Best Practices and Patterns

The design of an ensemble such as the swarm of garbage-collecting robots poses many difficult trade-offs and design decisions for the developer: What capabilities should each robot have, and is it better to use many simple, inexpensive robots, or would it be better to use a small number of larger, more powerful robots? Should the swarm be homogeneous or should it contain robots with specialized capabilities? What kind of awareness and knowledge do the robots need? How much knowledge do robots share, how do they assess the quality of the knowledge they acquire from sensors and other robots, and how should robots deal with contradictory information? Should robots have simple, predictable behaviors or more complex ones that have possibly greater potential for adaptation but also for unexpected failures? Should formal methods be used in the development process, and if so, which properties should be validated?

This is just a small selection of the high-level design decisions that have to be taken; while the system is developed and maintained, countless alternatives and design choices, at various levels of detail, have to be evaluated. This will always remain a challenging task that requires experience and domain knowledge on the part of the designer. But best practices can help designers to ask the right questions, to consider the problems that might arise in depth, and to evaluate the various trade-offs involved in different solutions as objectively as possible.

In the development of traditional software, and in particular in the area of distributed systems, patterns [9,10] have proven to be a valuable contribution. In general terms, an analysis or design pattern is a reusable solution to a development problem that specifies the compromises required by the solution as well as

its influence on other, related development problems. Pattern libraries provide a uniform vocabulary that simplifies the discussion of design choices, and they are repositories of proven solutions to common design problems.

In ASCENS we want to expand the pattern-based approach to include patterns for key features and mechanisms of SCs and SCEs (*adaptation, awareness, knowledge*, and *emergence*) at different levels of abstraction. An example is [24] which includes patterns that help designers to move from "black-box" descriptions (what adaptation, awareness, knowledge and emergence should achieve) to "white-box" solutions (how adaptation, awareness, knowledge and emergence can be realized). In this taxonomy, the robots of our simple case study are instances of the "reactive component" pattern and the ensemble follows the "environment mediated swarm intelligence" pattern. In the long term our goal is to provide a semi-formal language for our patterns that allows better integration of the pattern catalog into the ASCENS software development environment. A formal representation of patterns might even enable SCEs to reason about, e.g., structural patterns at run time, and hence use the pattern catalog to autonomously adapt the internal structure of the ensemble.

8.2 Awareness- and Knowledge-Cognizant Software Engineering

The SCEL model of the garbage-collecting robots in Sect. 6.2 is purely reactive, with little awareness of the environment and simple behaviors of the individual robots. While an ensemble built from very simple components may be sufficient in some scenarios, there are many cases where more complex behaviors are required. If we look at a more realistic version of the garbage-collecting robots, they will have to navigate in a complex environment, in which they have not only to avoid collisions with humans, they have to do so in an acceptable manner—driving at full speed in the direction of a visitor and then turning to avoid a collision at the last moment is simply not acceptable. Similarly; the robots have to distinguish garbage from other objects—they should, for example, definitely not remove the exhibits. To this end they may need the capability to learn, e.g., by driving around the exhibition hall before the opening in order to learn which objects belong to the exhibition. To fulfill these tasks the robots will need much more awareness, knowledge and reasoning capabilities than the simple system presented in Sect. 6. In ASCENS we are investigating a development approach for these kinds of system based on the foundations presented in this paper and inspired by previous work in the areas of artificial intelligence and multi-agent systems.

One of the important ingredients of this process will be a set of patterns for awareness- and knowledge-intensive components and ensembles. These patterns will specify the consequences and trade-offs for different ways of gathering and maintaining the data for awareness, and different processes of turning raw data into knowledge that can be used in the development process or while the system is executing. To support the developer beyond the purely conceptual stages of development, we are designing and implementing the Pseudo-Operational Ensemble Modeling Language (Poem). Poem is a specification language for

behavior and goals. It includes support for logical reasoning about fluents and modeling with relational Markov decision processes; Poem models can contain SCEL programs to describe executable behaviors.

8.3 Tool Support

Developing ensembles forces designers to deal with a multitude of languages, platforms, and tools. These concerns are also present in more traditional software development, but they are aggravated by the increased focus on awareness, knowledge and adaptation when developing ensembles.

Therefore we are developing a Software Development Environment (SDE) that integrates the various tools needed for modeling, validating, deploying and monitoring ensembles. The SDE has its origin in the SENSORIA project [23,17]; it is based on the Eclipse platform [7] and its underlying OSGi [18] framework. The core of the SDE allows for a straightforward integration of tools as well as the creation and use of tool chains built as orchestration of tools. Creating a new service as an orchestration of existing services is possible using either a textual, JavaScript-based approach or a graphical workflow approach.

As an example, a tool chain could be defined in the SDE consisting of a modeling tool for the specification of the swarm of robots described in the introduction, a tool for steady-state ODE simulation, and the ARGoS simulator for robot swarms. The developer can then define an orchestration of these tools that, e.g., generates traces from simulation runs and use them to validate the quantitative ODE models.

9 Concluding Remarks

In this paper we have presented some of the first results of the ASCENS systematic engineering of autonomic service-component ensembles. We have given short introductions to the SOTA approach to ensemble engineering and the underlying formal model called GEM, formal notions of adaptation and awareness, the SCEL language, quantitative analysis of ensembles, and finally envisaged software-engineering methods for ensembles.

But these results represent only a small part of the ASCENS project. In addition, the ASCENS project is developing an knowledge representation language, called KnowLang [21], for modelling four different types of knowlegde: the knowledge of the service components, the knowledge of the ensemble, context knowledge and situational knowledge. Validation and verification techniques in ASCENS are not restricted to quantitative model analysis; we also investigate qualitative model analysis (see e.g. [2]), runtime monitoring (see e.g. [8]), predictive analysis, the correspondence between the models and the implementation, and implementation-specific issues not covered by the models.

Particular emphasis is put on case studies. The one in this paper shows only a small part of our swarm-robotics approach which aims at ensembles of cooperating, self-aware robots. The Science Cloud case study is about making cloud

computing more dynamic and open while attempting to maintain its properties of being a reliable and flexible approach for using third-party resources and services. The e-mobility case study aims at illustrating the theories and methodologies developed in ASCENS in the domain of e-mobility planning.

The case studies provide not only continuous feedback to the research performed in ASCENS, they also lead to new scientific results in the case study domains (see e.g. [19,6,1]) and help to achieve the overall aim of ASCENS: a unified development approach to build self-aware, self-adaptive and self-expressive systems that can operate in open-ended, non-deterministic environments, perform in a reliable, predictable manner, adapt to changing environments or requirements, and handle failures of individual nodes.

Acknowledgements. This work has been partially sponsored by the FET-IST project FP7-257414 ASCENS. We thank Annabelle Klarl for reading drafts of the paper and useful comments.

References

1. Abeywickrama, D.B., Zambonelli, F.: Model checking goal-oriented requirements for self-adaptive systems. In: Popovic, M., Schätz, B., Voss, S. (eds.) ECBS, pp. 33–42. IEEE (2012)
2. Bensalem, S., Griesmayer, A., Legay, A., Nguyen, T.H., Peled, D.: Efficient deadlock detection for concurrent systems. In: Singh, S., Jobstmann, B., Kishinevsky, M., Brandt, J. (eds.) MEMOCODE, pp. 119–129. IEEE (2011)
3. Bruni, R., Corradini, A., Gadducci, F., Lluch-Lafuente, A., Vandin, A.: A conceptual framework for adaptation. In: de Lara, Zisman (eds.) [16], pp. 240–254
4. De Nicola, R., Ferrari, G.L., Pugliese, R.: Klaim: A kernel language for agents interaction and mobility. IEEE Trans. Software Eng. 24(5), 315–330 (1998)
5. De Nicola, R., Ferrari, G., Loreti, M., Pugliese, R.: A Language-Based Approach to Autonomic Computing. In: Beckert, B., de Boer, F., Bonsangue, M., Damiani, F. (eds.) FMCO 2011. LNCS, vol. 7542, pp. 25–48. Springer, Heidelberg (2012)
6. Eckhardt, J., Mühlbauer, T., AlTurki, M., Meseguer, J., Wirsing, M.: Stable availability under denial of service attacks through formal patterns. In: de Lara, Zisman (eds.) [16], pp. 78–93
7. Eclipse Foundation: The Eclipse Open Source Community and Java IDE (2011), `http://www.eclipse.org/` (accessed: August 02, 2012)
8. Falcone, Y., Jaber, M., Nguyen, T.H., Bozga, M., Bensalem, S.: Runtime Verification of Component-Based Systems. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 204–220. Springer, Heidelberg (2011)
9. Fowler, M.: Analysis Patterns: Reusable Object Models. Addison-Wesley Longman, Amsterdam (1996)
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley, Boston (1995)
11. Hillston, J., Tribastone, M., Gilmore, S.: Stochastic process algebras: From individuals to populations. Comput. J. 55(7), 866–881 (2012)
12. Hölzl, M., Rauschmayer, A., Wirsing, M.: Engineering of Software-Intensive Systems: State of the Art and Research Challenges. In: Wirsing, M., Banâtre, J.-P., Hölzl, M., Rauschmayer, A. (eds.) Software-Intensive Systems. LNCS, vol. 5380, pp. 1–44. Springer, Heidelberg (2008)

13. Hölzl, M., Wirsing, M.: Towards a System Model for Ensembles. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 241–261. Springer, Heidelberg (2011)
14. InterLink Project: Website, `http://interlink.ics.forth.gr/central.aspx` (accessed: August 02, 2012)
15. Keeney, R., Raiffa, H.: Decisions with multiple objectives: Preferences and value tradeoffs. J. Wiley, New York (1976)
16. de Lara, J., Zisman, A. (eds.): FASE 2012. LNCS, vol. 7212. Springer, Heidelberg (2012)
17. Mayer, P., Ráth, I.: The Sensoria Development Environment. In: Wirsing, Hölzl (eds.) [22], pp. 622–639
18. OSGi Alliance: OSGi Specification Release 4 (March 2008), `http://www.osgi.org/Specifications/` (accessed: August 02, 2012)
19. Pinciroli, C., Trianni, V., O'Grady, R., Pini, G., Brutschy, A., Brambilla, M., Mathews, N., Ferrante, E., Caro, G.D., Ducatelle, F., Stirling, T.S., Gutiérrez, Á., Gambardella, L.M., Dorigo, M.: ARGoS: A modular, multi-engine simulator for heterogeneous swarm robotics. In: IROS, pp. 5027–5034. IEEE (2011)
20. Russell, S.J., Norvig, P.: Artificial Intelligence - A Modern Approach (3rd internat. edn.). Pearson Education (2010)
21. Vassev, E., Hinchey, M., Gaudin, B., Nixon, P.: Requirements and Initial Model for KnowLang – a Language for Knowledge Representation in Autonomic Service-Component Ensembles. In: C3S2E 2011: The Fourth International C* Conference on Computer Science & Software Engineering, pp. 35–42. ACM (2011)
22. Wirsing, M., Hölzl, M. (eds.): Sensoria Project. LNCS, vol. 6582. Springer, Heidelberg (2011)
23. Wirsing, M., Hölzl, M.M., Koch, N., Mayer, P.: Sensoria - Software Engineering for Service-Oriented Overlay Computers. In: Wirsing, Hölzl (eds.) [22], pp. 1–14
24. Zambonelli, F., Bicocchi, N., Cabri, G., Leonardi, L., Puviani, M.: On Self-Adaptation, Self-Expression and Self-Awareness for Autonomic Service Component Ensembles. In: Proceedings of the 1st SASO Workshop on Self-Awareness, Ann Arbor, USA, pp. 108–113. IEEE CS Press (October 2011)

A Language-Based Approach to Autonomic Computing*

Rocco De Nicola[1], Gianluigi Ferrari[2], Michele Loreti[3], and Rosario Pugliese[3]

[1] IMT, Institute for Advanced Studies Lucca, Italy
[2] Università degli Studi di Pisa, Italy
[3] Università degli Studi di Firenze, Italy

Abstract. SCEL is a new language specifically designed to model autonomic components and their interaction. It brings together various programming abstractions that permit to directly represent knowledge, behaviors and aggregations according to specific policies. It also supports naturally programming self-awareness, context-awareness, and adaptation. In this paper, we first present design principles, syntax and operational semantics of SCEL. Then, we show how a dialect can be defined by appropriately instantiating the features of the language we left open to deal with different application domains and use this dialect to model a simple, yet illustrative, example application. Finally, we demonstrate that adaptation can be naturally expressed in SCEL.

1 Introduction

The increasing complexity, heterogeneity and dynamism of current computational and information infrastructures is calling for new ways of designing and managing computer systems and applications. *Adaptation*, namely "the capability of a system to change its behavior according to new requirements or environment conditions" [1], has been largely proposed as a powerful means for taming the ever-increasing complexity of today's computer systems and applications. Besides, a new paradigm, named *autonomic computing* [2], has been advocated that aims at making modern distributed IT systems *self-manageable*, i.e. capable of continuously self-monitoring and selecting appropriate operations.

More recently, to capture the relevant features and challenges, the 'Interlink WG on software intensive systems and new computing paradigms' [3] has proposed to use the term *ensembles* to refer to:

> The future generation of software-intensive systems dealing with massive numbers of components, featuring complex interactions among components and with humans and other systems, operating in open and non-deterministic environments, and dynamically adapting to new requirements, technologies and environmental conditions.

* This work has been partially sponsored by the EU project ASCENS (257414).

B. Beckert et al. (Eds.): FMCO 2011, LNCS 7542, pp. 25–48, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Systems partially satisfying the above definition of ensemble have been already built, e.g. national infrastructures such as power grids, or large online cloud systems such as Amazon or Google. But significant human intervention is needed to dynamically adapt them. Instead, one crucial requirement is to ensure that an ensemble continues to function reliably in spite of unforeseen changes and that adaptation does not render systems inoperable, unsafe or insecure.

To move from the engineering of traditional systems to that of ensembles, an higher level of abstraction is needed. Many research efforts are currently devoted to the search of methodologies and tools to build ensembles by exploiting techniques developed in different research areas such as software engineering, artificial intelligence and formal methods. The aim is the definition of linguistic primitives and methodologies to program autonomic and adaptive systems while relying on rigorous foundations that support verification of their properties.

The challenge for language designers is to devise appropriate abstractions and linguistic primitives to deal with the large dimension of systems, the need to adapt to evolving requirements and to changes in the working environment, and the emergent behaviors resulting from complex interactions.

The notions of *service components* (SCs) and *service-component ensembles* (SCEs) have been put forward as a means to structure a system into well-understood, independent and distributed building blocks that interact in specified ways. SCs are autonomic entities that can cooperate, with different roles, in open and non-deterministic environments. SCEs are instead sets of SCs with dedicated knowledge units and resources, featuring goal-oriented execution. Most of the basic properties of SCs and SCEs are already guaranteed by current service-oriented architectures; the novelty lays in the notions of goal-oriented evolution and of self-awareness and context-awareness.

A possible way to achieve awareness is to equip SCs and SCEs with information about their own state and behavior, to enable them to collect and store information about their working environment and to use this information for redirecting and adapting their behavior. A typical SCE is reported in Figure 1, which evidences that ensembles are structured sets of components, with dedicated *knowledge units* to represent shared, local and global knowledge, that can be interconnected via highly dynamic *infrastructures*.

These notions of SCs and SCEs are the starting point of the EU project ASCENS [4,5] that aims at investigating different issues ranging from languages for modelling and programming SCEs to foundational models for adaptation, dynamic self-expression and reconfiguration, from formal methods for the design and verification of reliable SCEs to software infrastructures supporting deployment and execution of SCE-based applications. The aim is to develop formal tools and methodologies supporting the design of self-adaptive systems that can autonomously adapt to, also unexpected, changes in the operating environment, while keeping most of their complexity hidden from administrators and users.

In this paper we present some of the work done to develop linguistic supports for modelling and programming service components and their ensembles. More specifically, we introduce SCEL (Service Component Ensemble Language), a

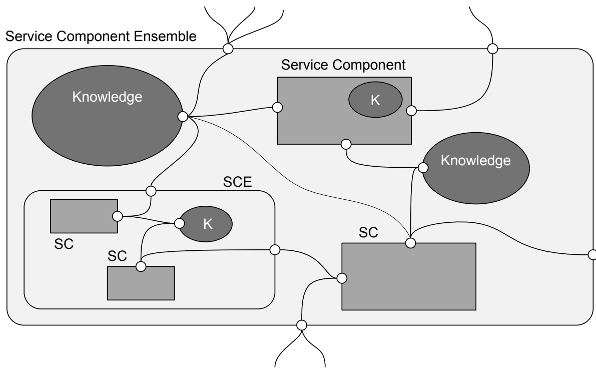


Fig. 1. A Service Component Ensemble

new language designed for autonomic computing. SCEL brings together different programming abstractions that permit to directly represent *knowledge*, *behaviors* and *aggregations* according to specific *policies*. It also supports naturally programming self-awareness, context-awareness and adaptation. SCEL's solid semantic grounds lay the basis for developing logics, tools and methodologies for formal reasoning about system behavior to establish qualitative and quantitative properties of both the individual components and the ensembles.

The rest of the paper is organized as follows. We present SCEL's design principles, syntax and operational semantics in Sections 2, 3 and 4, resp.. In Section 5, we show an example of how a dialect of SCEL can be defined by appropriately instantiating the features of the language we left open to deal with different application domains and in Section 6 we demonstrate how the proposed dialect can be used to model a simple yet illustrative example application. In Section 7 we argue that adaptation can be naturally expressed in SCEL. We review more strictly related work in Section 8 and conclude in Section 9 with some final remarks and hints for future work.

2 SCEL: Design Principles

SCEL provides abstractions explicitly supporting autonomic computing systems in terms of *Aggregations*, *Behaviors* and *Knowledge* according to specific *Policies*.

Aggregation Abstractions describe how different entities are brought together to form *components* and *systems* and to offer the possibility to construct the *software architecture* of autonomic systems. Composition of components and their interaction is implemented by exploiting the notion of *interface* that can be queried to determine the attributes and the functionalities provided and/or required by components. *Ensembles* are specific aggregations of components that represent *social or technical networks* of autonomic components. The key point is that the formation rule is endogenous to components: components of an ensemble are connected by the interdependency relations established in their interfaces. Therefore, an ensemble is not a rigid fixed network but rather a dynamic graph-like structure where component linkages are dynamically established.

Behavioral Abstractions describe how computations progress. These abstractions are modelled as processes in the style of standard process calculi. *Interaction* comes in when components access data in the knowledge repositories of other components. *Adaptation* emerges as the result of knowledge acquisition and manipulation.

Knowledge Abstractions provide the high level primitives to manage pieces of relevant information coming from different sources. Knowledge is represented through items stored in repositories. Knowledge items contain either *application data* or *awareness data*. The former are used for determining the progress of component computations, while the latter provide information about the environment in which the different components are running (e.g. monitored data from sensors) or about the actual status of an autonomic component (e.g. about its current position or the remaining battery's charge level). We assume that each knowledge repository provides three abstract operations that can be used by autonomic components for *adding* new knowledge to the repository, for *retrieving* knowledge from the repository and for *withdrawing* knowledge from it.

Policy Abstractions deal with the way behaviors are regulated. Since few assumptions can be made about the operational environment, that is frequently open, highly dynamic and possibly hostile, the ability of programming and enforcing a finer control on behavior is essential to assure that valuable information is not lost. Policies are the mean to guarantee such control. Interaction policies and Service Level Agreement (SLA) policies provide two standard examples of policy abstractions. Other examples are security properties maintaining the right linkage between data values and their associated usage policies (data-leakage policies) or limiting the flow of sensitive information to untrusted sources (access control and reputation policies).

The two central ingredients of SCEL are the notions of *autonomic component* and of *ensemble* that we shall additionally consider below.

2.1 Components

An *autonomic component* $\mathcal{I}[\mathcal{K}, \Pi, P]$ consists of:

1. an *interface* $\mathcal{I}$ publishing and making available structural and behavioral information about the component itself;
2. a *knowledge manager* $\mathcal{K}$, managing both application data and awareness data, together with the specific handling mechanism;
3. a set of *policies* Π regulating the interaction between the different internal parts of the component and the interaction of the component with the others;
4. a *process* P together with a set of process definitions that can be dynamically activated. Some of the processes in P perform local computation, while others may coordinate processes interaction with the knowledge repository and deal with the issues related to adaptation and reconfiguration.

Component interfaces can be inquired to extract components name, the interdependencies among components, and the services offered by components. Indeed, the interface of a component provides at least the following attributes:

- *id*: its name;
- *ensemble*: a predicate on interfaces used to determine the ensemble the component has created and currently coordinates;
- *membership*: a predicate on the interfaces used to determine the ensembles which the component is willing to be member of.

Additional attributes might, e.g., indicate the battery's charge level and the component's GPS position.

Notably, the whole information provided by the component interface is stored in the local knowledge of the component and therefore it can be dynamically changed by using the appropriate operators for knowledge handling.

2.2 Ensembles

Ensembles are aggregations of components characterized by means of suitable predicates associated to the attributes *ensemble* and *membership*. Surprisingly (it might be), no specific syntactic category or operator for forming ensembles is provided by SCEL. Rather, to better capture their dynamicity, ensembles are 'synthesized' dynamically by exploiting the values of the components attributes. This design choice guarantees high dynamicity and flexibility in forming, joining and leaving ensembles and does avoid resorting to structure ensembles through rigid syntactic constructs.

For example, the names of the components that can be members of an ensemble can be fixed via the predicate

$$P(\mathcal{I}) \stackrel{def}{=} \mathcal{I}.id \in \{n, m, p\}$$

If the attribute *ensemble* of a component C has assigned $P(\mathcal{I})$, then a component C' is part of the ensemble coordinated by C if its name is n, m or p. Of course, the predicate assigned to *ensemble* can be changed dynamically, thus permitting to modify at run-time the members of the ensemble coordinated by C.

As another example, to dynamically characterize the members of an ensemble that are active and have a battery charge level greater than 30%, the predicate

$$P(\mathcal{I}) \stackrel{def}{=} \mathcal{I}.active = yes \wedge \mathcal{I}.battery_level > 30\%$$

could be used. Here, we are assuming that the interface of each component willing to be part of the ensemble contains the attributes *active* and *battery_level*.

Components, in turn, could be willing to be part of any ensemble. This is modelled by letting attribute *membership* be associated to the predicate *true*. On the contrary, components may not want to be part of any ensemble, in this case *membership* will be set to be *false*. More generally, components can place restrictions on the ensembles which they are willing to be member of by appropriately setting the attribute *membership*. For example, using the predicate

$$P(\mathcal{I}) \stackrel{def}{=} \mathcal{I}.trust_level > medium$$

a component can express its willingness to be only part of those ensembles coordinated by components whose (certified) trust level is greater than medium.

Table 1. SCEL syntax ($\mathcal{K}$, Π, T, and t are parameters)

Systems:
$S ::= C \mid S_1 \parallel S_2 \mid (\nu n)S$

Components:
$C ::= \mathcal{I}[\mathcal{K}, \Pi, P]$

Processes:
$P ::= \mathbf{nil} \mid a.P \mid P_1 + P_2 \mid P_1[\,P_2\,] \mid X \mid A(\bar{p}) \quad (A(\bar{f}) \triangleq P)$

Actions:
$a ::= \mathbf{get}(T)@c \mid \mathbf{qry}(T)@c \mid \mathbf{put}(t)@c \mid \mathbf{exec}(P) \mid \mathbf{new}(\mathcal{I}, \mathcal{K}, \Pi, P)$

Targets:
$c ::= n \mid x \mid \mathsf{self}$

3 SCEL: Syntax

The syntax of SCEL is illustrated in Table 1. There, different syntactic categories are defined that constitute the main ingredients of our language. The basic category of the syntax is that relative to PROCESSES that are used to build up COMPONENTS that in turn are used to define SYSTEMS. PROCESSES model the flow of the ACTIONS that can be performed. Each ACTION has among its parameters a TARGET, that indicates the other component that is involved in that action, and either an ITEM or a TEMPLATE, that determines the part of KNOWLEDGE to be added, retrieved or removed. POLICIES are used to control and adapt the actions of the different components in order to guarantee the achievement of specific goals or the satisfaction of specific properties.

It has to be said that our aim is to identify linguistic constructs for uniformly modeling the control of computation, the interaction among possibly heterogeneous components, and the architecture of systems and ensembles. Therefore, we have left open some syntactic categories, namely Knowledge (ranged over by $\mathcal{K}$), Policies (Π), Templates (T), and Items (t). These represent additional language features that need to be introduced, e.g. to represent and store knowledge of different forms (e.g. constraints, clauses, records, tuples) or to express a variety of policies (e.g. to regulate knowledge handling, resource usage, process execution, process interaction, actions priority, security, trust, reputation). We don't want to take a definite standing about these categories and prefer they be fixed from time to time according to the specific application domain or to the taste of the language user. In the rest of this section, we consider one by one the explicitly defined categories and describe them in detail.

Processes are the SCEL active computational units. Each process is built up from the *inert* process **nil** via *action prefixing* ($a.P$), *nondeterministic choice* ($P_1 + P_2$), *controlled composition* ($P_1[\,P_2\,]$), *process variable* (X), *parameterised process invocation* ($A(\bar{p})$), and *parameterised process definition* ($A(\bar{f}) \triangleq P$). The construct $P_1[\,P_2\,]$ abstracts the various forms of parallel composition commonly used in process calculi. Process variables are used to support higher-order communication, namely the capability to exchange (the code of) a process by first adding an item containing the process to a knowledge repository and then retrieving/withdrawing this item while binding the process to a process variable.

Processes can perform five different kinds of ACTIONS. Actions $\mathbf{get}(T)@c$, $\mathbf{qry}(T)@c$ and $\mathbf{put}(t)@c$ are used to manage shared knowledge repositories by withdrawing/retrieving/adding information items from/to the knowledge repository c. These operations exploit templates T as patterns to select knowledge items t in the repositories. They rely heavily on the used knowledge repository and are implemented by invoking the handling operations it provides. Action $\mathbf{exec}(P)$ triggers a controlled (parallel) execution of process P. Action $\mathbf{new}(\mathcal{I}, \mathcal{K}, \Pi, P)$ creates a new component $\mathcal{I}[\mathcal{K}, \Pi, P]$.

Action **get** is a *blocking* action, in the sense that the process executing it has to wait for the wanted element if it is not (yet) available in the knowledge repository. Action **qry**, exactly like **get**, suspends the process executing it if the knowledge repository does not (yet) contain or cannot 'produce the wanted element. The two blocking actions differ also for the fact that **get** removes the found item from the knowledge repository while **qry** leaves the target repository unchanged. Actions **put**, **exec** and **new** are instead non-blocking and are immediately executed.

Component names are denoted by n, n', ..., variables for names are denoted by x, x', ..., while c stands for a name or a variable. The distinguished variable self can be used by processes to refer to the name of their hosting component.

SYSTEMS aggregate COMPONENTS (see Section 2.1) through the *composition* operator $_ \parallel _$. It is also possible to restrict the scope of a name, say n, by using the *name restriction* operator $(\nu n)_$. Thus, in a system of the form $S_1 \parallel (\nu n)S_2$, the effect of the operator is to make name n invisible from within S_1. Essentially, this operator plays a role similar to that of a *begin ... end* block in sequential programming and limits visibility of specific names. Additionally, it allows components to communicate restricted names thus enlarging their scope to encompass also the receiving components (like restriction in π-calculus [6]).

4 SCEL: Operational Semantics

The operational semantics of SCEL is given in the SOS style [7] by relying on the notion of Labelled Transition System (LTS), which is a triple $\langle \mathcal{S}, \mathcal{L}, \longrightarrow \rangle$ made of a set of states $\mathcal{S}$, a set of transition labels $\mathcal{L}$, and a labelled transition relation $\longrightarrow \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ accounting for the actions that can be performed from each state and the new state reached after each such transition. The semantics is defined in two steps: first, the semantics of processes specifies process commitments ignoring process allocation, available data, regulating policies, etc.; then, by taking process commitments and system configuration into account, the semantics of systems provides a full description of systems behavior.

To define the semantics, we use the sets of *bound* variables $bv(E)$ and *free* variables $fv(E)$, and the sets of names $n(E)$, *bound* names and *free* names occurring in a syntactic term E. These sets, as usual, can be defined inductively on the syntax of actions, processes, components, and systems by taking into account that the only binding constructs are actions **get** and **qry** as concerns variables and action **new** and the restriction operator as concerns names. More precisely,

Table 2. Operational semantics of processes

$$a.P \xmapsto{a} P \quad (a \neq \mathbf{exec}(Q)) \qquad \mathbf{exec}(Q).P \xmapsto{\mathbf{exec}(Q)} P[\,Q\,] \qquad P \xmapsto{\circ} P$$

$$\frac{P \xmapsto{\alpha} P'}{P + Q \xmapsto{\alpha} P'} \qquad \frac{Q \xmapsto{\alpha} Q'}{P + Q \xmapsto{\alpha} Q'} \qquad \frac{A(\bar{f}) \triangleq P \quad P\{\bar{p}/\bar{f}\} \xmapsto{\alpha} P'}{A(\bar{p}) \xmapsto{\alpha} P'}$$

$$\frac{P \xmapsto{\alpha} P' \quad Q \xmapsto{\beta} Q'}{P[\,Q\,] \xmapsto{\alpha[\,\beta\,]} P'[\,Q'\,]}\ bv(\alpha) \cap bv(\beta) = \emptyset \qquad \frac{P =_\alpha P' \quad P' \xmapsto{\beta} P''}{P \xmapsto{\beta} P''}$$

actions $\mathbf{get}(T)@c$ and $\mathbf{qry}(T)@c$ bind the variables occurring in the template T, while action $\mathbf{new}(\mathcal{I}, \mathcal{K}, \Pi, P)$ binds the name associated to attribute $\mathcal{I}.id$; the scope of these binders is the process P_1 syntactically following the action in a prefix form $a.P_1$. The restriction operator $(\nu n)_$ binds n in the scope $_$. A term without free variables is deemed *closed* (it may contain free names).

The semantics is only defined for closed systems. Indeed, we consider the binding of a variable as its declaration (and initialisation), therefore free occurrences of variables at the outset in a system must be prevented since they are similar to uses of variables before their declaration in programs (which are considered as programming errors).

4.1 Operational Semantics of Processes

The semantics of processes specifies process commitments, i.e. the actions that processes can initially perform. That is, given a process P, its semantics points out all the actions that P can initially perform and the continuation process P' obtained after each such action. To simplify the rules, we do not restrict them (and the semantics) to the subset of closed processes, although when defining the semantics of systems we only consider the transitions from closed processes (see Section 4.2). Moreover, we only consider processes that are such that their bound names are pairwise distinct and different from their free names.

The LTS defining the semantics of processes is given as follows:

- the set of states coincides with the set of processes as defined in Table 1;
- the set of transition labels is generated by the following production rule

  $$\alpha, \beta ::= a \;\big|\; \circ \;\big|\; \alpha[\,\beta\,]$$

  meaning that a label is either an action as defined in Table 1, or the symbol $\circ$, denoting inaction, or the composition $\alpha[\,\beta\,]$ of two labels α and β;
- the labelled transition relation $\longmapsto$ is the least relation induced by the inference rules in Table 2. We will use P and Q, possibly indexed, to range over processes and write $P \xmapsto{\alpha} Q$ instead of $\langle P, \alpha, Q \rangle \in \longmapsto$.

The rules defining the labelled transition relation are straightforward. In particular, **exec** spawns a new concurrent process whose execution can be controlled

by the continuation of the process performing the action. The rule defining the semantics of $P[\,Q\,]$ states that a transition labeled $\alpha[\,\beta\,]$ is performed when Q makes the action β while P makes the action α. However, P and Q are not forced to synchronise. Indeed, thanks to the third rule, that allows any process to perform a $\circ$-labelled transition, α and/or β may always be $\circ$. The semantics of $P[\,Q\,]$ at the level of processes is indeed absolutely permissive and generates all possible compositions of the commitments of P and Q. This semantics is then specialized at the level of systems by means of interaction predicates for taking policies into account. Condition $bv(\alpha) \cap bv(\beta) = \emptyset$ means that the variables freed by the action $\alpha[\,\beta\,]$ in the two processes P and Q must be different: this because they correspond to bound variables that were intended to be different (although they might have had the same identity) and, once they get free, could be subject to possibly different substitutions (substitutions are generated and applied by rule *(pr-sys)* in Table 3). Notably, also this condition is not strict: it can be always made true by application of the last rule stating that *α-equivalent processes*, i.e. processes only differing in the identity of bound variables (this equivalence relation is denoted by $=_\alpha$), perform the same transitions.

4.2 Operational Semantics of Systems

The operational semantics of systems is defined in two steps. First, we define an LTS to derive the transitions enabled from systems without occurrences of the name restriction operator.Then, by exploiting this LTS, we provide the semantics of generic systems by means of a (unlabelled) transition system (TS), that is a pair $\langle \mathcal{S}, \rightarrowtail \rangle$ made of a set of states $\mathcal{S}$ and a (unlabelled) transition relation $\rightarrowtail \subseteq \mathcal{S} \times \mathcal{S}$ accounting for the computation steps that can be performed from each state and the new state reached after each such transition. This approach allows us to avoid the intricacies, also from a notational point of view, arising when dealing with name mobility in computations (e.g. when opening and closing the scopes of name restrictions).

To simplify notation, we will use $\mathcal{I}$ and $\mathcal{J}$ to range over interfaces. Notation $\mathcal{I} \models \mathcal{J}.ensemble$ indicates that a component with interface $\mathcal{J}$ is willing to accept a component with interface $\mathcal{I}$ in the ensemble it coordinates. Similarly, $\mathcal{J} \models \mathcal{I}.membership$ indicates that $\mathcal{I}$ is willing to be one of the components of the ensemble coordinated by $\mathcal{J}$. We assume that it always implicitly holds that $\mathcal{I} \models \mathcal{I}.ensemble \ \wedge \ \mathcal{I} \models \mathcal{I}.membership$, i.e. that a component is always part of the ensemble it coordinates. Moreover, we assume that the names of the attributes of a component are just pointers to the actual values contained in the knowledge repository associated to the component. This amounts to saying that in terms of the form $\mathcal{I}[\mathcal{K}, \Pi, P]$, $\mathcal{I}$ only includes the names of the attributes, as their corresponding values can be easily retrieved from $\mathcal{K}$. However, when $\mathcal{I}$ is used in isolation, it also includes the attributes' values.

The LTS defining the semantics of systems without restricted names is $\langle \mathcal{S}, \mathcal{L}, \longrightarrow \rangle$ where

- the set of states $\mathcal{S}$ includes all the systems defined in Table 1;
- $\mathcal{L}$ is the set of transition labels generated by the following production rule

$$\lambda ::= \tau \mid \mathcal{I} : \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, P) \mid \mathcal{I} \diamond \mathcal{J} \mid \mathcal{I} : t \lhd c \mid \mathcal{I} : t \blacktriangleleft c \mid \mathcal{I} : t \rhd c \\ \mid \mathcal{I} : t \,\bar{\lhd}\, \mathcal{J} \mid \mathcal{I} : t \,\bar{\blacktriangleleft}\, \mathcal{J} \mid \mathcal{I} : t \,\bar{\rhd}\, \mathcal{J}$$

  where τ denotes an internal computation step, $\mathcal{I} : \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, P)$ denotes the willingness of component $\mathcal{I}$ to create the new component $\mathcal{J}[\mathcal{K}, \Pi, P]$, $\mathcal{I} \diamond \mathcal{J}$ denotes the willingness of two components with interfaces $\mathcal{I}$ and $\mathcal{J}$ to interact, $\mathcal{I} : t \lhd c$ ($\mathcal{I} : t \blacktriangleleft c$) denotes the intention of component $\mathcal{I}$ to withdraw (retrieve) item t from the repository at c, $\mathcal{I} : t \rhd c$ denotes the intention of component $\mathcal{I}$ to add item t to the repository at c, $\mathcal{I} : t \,\bar{\lhd}\, \mathcal{J}$ ($\mathcal{I} : t \,\bar{\blacktriangleleft}\, \mathcal{J}$) denotes that component $\mathcal{I}$ is allowed to withdraw (retrieve) item t from the repository of component $\mathcal{J}$, $\mathcal{I} : t \,\bar{\rhd}\, \mathcal{J}$ denotes that component $\mathcal{I}$ is allowed to add item t to the repository of component $\mathcal{J}$;
- $\longrightarrow$ is the labelled transition relation induced by the inference rules in Table 3. We will write $S \xrightarrow{\lambda} S'$ instead of $\langle S, \lambda, S' \rangle \in \longrightarrow$.

The labelled transition relation relies on the following two predicates:

- the *interaction predicate*, $\Pi, \mathcal{I} : \alpha \succ \lambda, \sigma$, means that under policy Π and interface $\mathcal{I}$, process label α yields system label λ and substitution σ;
- the *authorization predicate*, $\Pi, \mathcal{I} \vdash \lambda$, means that under policy Π and interface $\mathcal{I}$, system label λ is allowed.

The interaction predicate establishes a relation between process labels and system labels and thus determines the system label λ to exhibit and the substitution σ to apply when a process performs a transition labeled α. It is called interaction predicate because its main role is determining the effect of the concurrent execution of different actions by different processes that, e.g., exhibit labels of the form $\alpha_1[\,\alpha_2\,]$. Many different interaction predicates can thus be defined to capture well-known process computation and interaction patterns such as interleaving, asynchronous communication, synchronous communication, full synchrony, broadcasting, etc. Despite the several interaction predicates that can be defined, we expect anyway that a well-defined interaction predicate satisfies some obvious criteria. For example, a process label of the form $\mathbf{get}(T)@c$ should be related to system labels of the form $\mathcal{I} : t \lhd c$, where t is any item 'matching' the template T, while a process label of the form $\mathbf{put}(t)@c$ should be related to system labels of the form $\mathcal{I} : t' \rhd c$, where t' is any item resulting from the evaluation of t. We refer the reader to [8] for some notable examples.

The authorization predicate is used to determine the actions that can be performed according to specific policies. Likewise the interaction predicate, many different reasonable authorization predicates can be defined depending on Π.

The labeled transition relation also relies on the following three operations that each knowledge repository's handling mechanism must provide:

Table 3. Semantics of systems: labelled transition relation (symmetric of rules *(syncget)*, *(syncqry)*, *(syncput)*, *(enscomm)* and *(async)* omitted)

$$\frac{P \overset{\alpha}{\longmapsto} P' \qquad \Pi, \mathcal{I} : \alpha \succ \lambda, \sigma}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\lambda} \mathcal{I}[\mathcal{K}, \Pi, P'\{\sigma\}]} \ \textit{(pr-sys)}$$

$$\frac{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:\mathbf{new}(\mathcal{J},\mathcal{K}',\Pi',P')} C \qquad n = \mathcal{J}.id \qquad n \notin n(\mathcal{I}[\mathcal{K}, \Pi, \mathbf{nil}])}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\tau} (\nu n)(C \parallel \mathcal{J}[\mathcal{K}', \Pi', P'])} \ \textit{(newc)}$$

$$\frac{\mathcal{K} \ominus t = \mathcal{K}' \qquad \Pi, \mathcal{I} \vdash \mathcal{I} : t\,\bar{\lhd}\,\mathcal{I} \qquad n = \mathcal{I}.id \qquad \mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:t \lhd n} \mathcal{I}[\mathcal{K}, \Pi, P']}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\tau} \mathcal{I}[\mathcal{K}', \Pi, P']} \ \textit{(lget)}$$

$$\frac{\mathcal{K} \ominus t = \mathcal{K}' \qquad \Pi, \mathcal{J} \vdash \mathcal{I} : t\,\bar{\lhd}\,\mathcal{J}}{\mathcal{J}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:t\,\bar{\lhd}\,\mathcal{J}} \mathcal{J}[\mathcal{K}', \Pi, P]} \ \textit{(accget)}$$

$$\frac{S_1 \xrightarrow{\mathcal{I}:t \lhd n} S_1' \qquad S_2 \xrightarrow{\mathcal{I}:t\,\bar{\lhd}\,\mathcal{J}} S_2' \qquad \mathcal{J}.id = n \qquad ens(\mathcal{I}, \mathcal{J}) \Rightarrow \lambda = \tau, \lambda = \mathcal{I} \diamond \mathcal{J}}{S_1 \parallel S_2 \xrightarrow{\lambda} S_1' \parallel S_2'} \ \textit{(syncget)}$$

$$\frac{\mathcal{K} \vdash t \qquad \Pi, \mathcal{I} \vdash \mathcal{I} : t\,\bar{\blacktriangleleft}\,\mathcal{I} \qquad n = \mathcal{I}.id \qquad \mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:t \blacktriangleleft n} \mathcal{I}[\mathcal{K}, \Pi, P']}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\tau} \mathcal{I}[\mathcal{K}, \Pi, P']} \ \textit{(lqry)}$$

$$\frac{\mathcal{K} \vdash t \qquad \Pi, \mathcal{J} \vdash \mathcal{I} : t\,\bar{\blacktriangleleft}\,\mathcal{J}}{\mathcal{J}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:t\,\bar{\blacktriangleleft}\,\mathcal{J}} \mathcal{J}[\mathcal{K}, \Pi, P]} \ \textit{(accqry)}$$

$$\frac{S_1 \xrightarrow{\mathcal{I}:t \blacktriangleleft n} S_1' \qquad S_2 \xrightarrow{\mathcal{I}:t\,\bar{\blacktriangleleft}\,\mathcal{J}} S_2' \qquad \mathcal{J}.id = n \qquad ens(\mathcal{I}, \mathcal{J}) \Rightarrow \lambda = \tau, \lambda = \mathcal{I} \diamond \mathcal{J}}{S_1 \parallel S_2 \xrightarrow{\lambda} S_1' \parallel S_2'} \ \textit{(syncqry)}$$

$$\frac{\mathcal{K} \oplus t = \mathcal{K}' \qquad \Pi, \mathcal{I} \vdash \mathcal{I} : t\,\bar{\rhd}\,\mathcal{I} \qquad n = \mathcal{I}.id \qquad \mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:t \rhd n} \mathcal{I}[\mathcal{K}, \Pi, P']}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\tau} \mathcal{I}[\mathcal{K}', \Pi, P']} \ \textit{(lput)}$$

$$\frac{\mathcal{K} \oplus t = \mathcal{K}' \qquad \Pi, \mathcal{J} \vdash \mathcal{I} : t\,\bar{\rhd}\,\mathcal{J}}{\mathcal{J}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:t\,\bar{\rhd}\,\mathcal{J}} \mathcal{J}[\mathcal{K}', \Pi, P]} \ \textit{(accput)}$$

$$\frac{S_1 \xrightarrow{\mathcal{I}:t \rhd n} S_1' \qquad S_2 \xrightarrow{\mathcal{I}:t\,\bar{\rhd}\,\mathcal{J}} S_2' \qquad \mathcal{J}.id = n \qquad ens(\mathcal{I}, \mathcal{J}) \Rightarrow \lambda = \tau, \lambda = \mathcal{I} \diamond \mathcal{J}}{S_1 \parallel S_2 \xrightarrow{\lambda} S_1' \parallel S_2'} \ \textit{(syncput)}$$

$$\frac{S \xrightarrow{\mathcal{I} \diamond \mathcal{J}} S' \qquad \mathcal{I} \in \mathcal{I}' \wedge \mathcal{J} \in \mathcal{I}' \qquad \Pi, \mathcal{I}' \vdash \mathcal{I} \diamond \mathcal{J}}{\mathcal{I}'[\mathcal{K}, \Pi, P] \parallel S \xrightarrow{\tau} \mathcal{I}'[\mathcal{K}, \Pi, P] \parallel S'} \ \textit{(enscomm)}$$

$$\frac{S_1 \xrightarrow{\lambda} S_1'}{S_1 \parallel S_2 \xrightarrow{\lambda} S_1' \parallel S_2} \ \textit{(async)}$$

- $\mathcal{K} \ominus t = \mathcal{K}'$: the *withdrawal* of item t from the repository $\mathcal{K}$ returns $\mathcal{K}'$;
- $\mathcal{K} \vdash t$: the *retrieval* of item t from the repository $\mathcal{K}$ is possible;
- $\mathcal{K} \oplus t = \mathcal{K}'$: the *addition* of item t to the repository $\mathcal{K}$ returns $\mathcal{K}'$.

Rule *(pr-sys)* transforms process labels into system labels by exploiting the interaction predicate $\Pi, \mathcal{I} : \alpha \succ \lambda, \sigma$. In particular, it generates the following system labels: τ, $\mathcal{I} : \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, P)$, $\mathcal{I} : t \triangleleft c$, $\mathcal{I} : t \blacktriangleleft c$ and $\mathcal{I} : t \triangleright c$. As a consequence of this transformation, a substitution σ (i.e. a function from variables to values) is generated and applied to the continuation of the process that has exhibited label α. This is necessary when α contains a **get** or a **qry**, because, due to the way the semantics of processes is defined, the continuation P' may contain free variables even if P is closed. It is worth noting that the domain of σ is the set of variables that are bound in α, thus, since $fv(P') \subseteq bv(\alpha)$, the process $P'\{\sigma\}$ is closed. The application of the rule also replaces self with the corresponding name.

No specific system label is used for indicating execution of action **exec**. Indeed, this action is always local to the component executing it, and no other component is involved in that action. Hence, when applying rule *(pr-sys)*, all the information (i.e. Π) needed to decide if the action can be allowed or not is present. When **exec** is allowed, the interaction predicate in the premise of the rule is of the form $\Pi, \mathcal{I} : \mathbf{exec}(Q) \succ \tau, []$, where $[]$ denotes the empty substitution, and the transition corresponds to an internal computation step.

Like the **exec**, action **new** is decided by using the information within a single component. However, since it affects the whole system as it creates a new component, its execution is indicated by a specific system label $\mathcal{I} : \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, P)$ (generated by rule *(pr-sys)*) carrying enough information for the creation of the new component to take place. When the new component is actually created *(newc)*, it is checked that its name n is not already used in the creating component possibly except for the process part (this condition can be always made true by exploiting α-equivalence among processes) and, if so, a restriction is put in the system obtained after the computation step to delimit the scope of n.

The successful execution of the remaining three actions requires, at system level, appropriate synchronizations. For this reason, for each action we have a pair of complementary labels. Action **get** withdraws an item either from the local repository, rule *(lget)*, or from a specific repository, rule *(syncget)*. In both cases, this transition corresponds to an internal computation step. However, in case of remote withdrawal, it is also needed to make sure that the interacting components belong to the same ensemble. We have two cases to consider, depending on predicate $ens(\mathcal{I}, \mathcal{J})$ defined as $(\mathcal{I} \models \mathcal{J}.ensemble \;\wedge\; \mathcal{J} \models \mathcal{I}.membership) \vee (\mathcal{J} \models \mathcal{I}.ensemble \;\wedge\; \mathcal{I} \models \mathcal{J}.membership)$:

- Predicate $ens(\mathcal{I}, \mathcal{J})$ holds true, i.e. the component with interface $\mathcal{I}$ is part of the ensemble defined by the component with interface $\mathcal{J}$, or viceversa. Then, the (conditional) premise $ens(\mathcal{I}, \mathcal{J}) \Rightarrow \lambda = \tau, \lambda = \mathcal{I} \diamond \mathcal{J}$ of rule *(syncget)* sets λ to τ and the inference of the computation step terminates.
- Predicate $ens(\mathcal{I}, \mathcal{J})$ holds false and the two components with interface $\mathcal{I}$ and $\mathcal{J}$ are both part of the ensemble coordinated by another component,

say $\mathcal{I}'[\mathcal{K}, \Pi, P]$. Indeed, we write $\mathcal{I} \in \mathcal{I}' \wedge \mathcal{J} \in \mathcal{I}'$ as a shorthand for condition $(\mathcal{I} \models \mathcal{I}'.ensemble \wedge \mathcal{I}' \models \mathcal{I}.membership) \wedge (\mathcal{J} \models \mathcal{I}'.ensemble \wedge \mathcal{I}' \models \mathcal{J}.membership)$. We now take advantage of the 'else' case of the premise $ens(\mathcal{I}, \mathcal{J}) \Rightarrow \lambda = \tau, \lambda = \mathcal{I} \diamond \mathcal{J}$ of rule *(syncget)* that sets λ to $\mathcal{I} \diamond \mathcal{J}$. Consequently, rule *(enscomm)* exploits the authorization predicate $\Pi, \mathcal{I}' \vdash \mathcal{I} \diamond \mathcal{J}$ to check whether the policy Π in force at $\mathcal{I}'$ authorizes interaction between $\mathcal{I}$ and $\mathcal{J}$ and, if so, infers the computation step.

The label $\mathcal{I} : t \,\bar{\triangleleft}\, \mathcal{J}$, generated by rule *(accget)*, denotes the willingness of a component $\mathcal{J}$ to provide t to a component $\mathcal{I}$. When $\mathcal{J}.id = n$, its complementary label is $\mathcal{I} : t \triangleleft n$ generated by rule *(pr-sys)* when a component $\mathcal{I}$ wants to withdraw t from the repository at n. When the target of the action denotes a remote repository, rule *(syncget)*, the action is only allowed if $\mathcal{J}.id = n$, namely if n is the name of the component with interface $\mathcal{J}$. The semantics of action **qry** is modelled by rules *(lqry)*, *(accqry)* and *(syncqry)*. This action behaves similarly to **get**, the only difference being that it invokes the retrieval operation of the repository's handling mechanism, rather than the withdrawal operation. Thus, if the action succeeds, the repository after the computation step remains unchanged. Action **put** adds item t to a repository. Its behavior is modelled by rules (namely *(lput)*, *(accput)* and *(syncput)*) similar to those of actions **get** and **qry**, the major difference being now that the addition operation of the repository's handling mechanism is invoked. In any case, for remote synchronisation to take place, it could require authorisation through the application of rule *(enscomm)*.

Finally, rule *(async)* allows a whole system to asynchronously evolve when only some of its components evolve.

It is worth noticing that, although the inference rules in Table 3 are defined on top of all the systems produced by the syntax in Table 1, no transition can be derived from a system containing name restrictions. That is, in a transition $S \xrightarrow{\lambda} S'$, S may not contain name restrictions (instead, because of rule *(newc)*, S' may do). This account for our statement at the beginning of this section, i.e. that we first define an LTS to derive the transitions enabled from systems without occurrences of name restrictions.

Now, the TS defining the semantics of generic systems is defined as

- the set of states $\mathcal{S}$ includes all the systems defined in Table 1;
- the transition relation $\rightarrowtail$ is the least relation induced by the inference rules in Table 4. As a matter of notation, we will write $S \rightarrowtail S'$ instead of $\langle S, S' \rangle \in \rightarrowtail$. Moreover, $\bar{n}$ denotes a (possibly empty) sequence of names and $\bar{n}, n'$ is the sequence obtained by composing $\bar{n}$ and n'. $(\nu \bar{n})S$ abbreviates $(\nu n_1)((\nu n_2)(\cdots(\nu n_m)S \cdots))$, if $\bar{n} = n_1, n_2, \cdots, n_m$, and S, otherwise. $S\{n'/n\}$ denotes the system obtained by replacing any free occurrence in S of n with n'. When considering a system S, a name is deemed *fresh* if it is different from any name occurring in S.

The rules defining the transition relation are straightforward. The first rule accounts for the computation steps of a system where all (possible) name restrictions are at top level, while the last two rules permit to manipulate the syntax of

Table 4. Semantics of systems: transition relation

$$\frac{S \xrightarrow{\tau} S'}{(\nu\bar{n})S \rightarrowtail (\nu\bar{n})S'} \ \textit{(res-tau)}$$

$$\frac{(\nu\bar{n}, n'')(S_1 \parallel S_2\{n''/n'\}) \rightarrowtail S' \quad n'' \ \textit{fresh}}{(\nu\bar{n})(S_1 \parallel (\nu n')S_2) \rightarrowtail S'} \ \textit{(res-top-r)}$$

$$\frac{(\nu\bar{n}, n'')(S_1\{n''/n'\} \parallel S_2) \rightarrowtail S' \quad n'' \ \textit{fresh}}{(\nu\bar{n})((\nu n')S_1 \parallel S_2) \rightarrowtail S'} \ \textit{(res-top-l)}$$

Table 5. Semantics of systems: inter-ensemble communication

$$\frac{S_1 \xrightarrow{\lambda_1} S_1' \qquad S_2 \xrightarrow{\lambda_2} S_2'}{S_1 \parallel S_2 \xrightarrow{\lambda_1 \diamond \lambda_2} S_1' \parallel S_2'} \ \textit{(ens1)}$$

$$\frac{S \xrightarrow{\lambda'} S' \qquad \Pi, \mathcal{I} \vdash \lambda' \succ \lambda}{\mathcal{I}[\mathcal{K}, \Pi, P] \parallel S \xrightarrow{\lambda} \mathcal{I}[\mathcal{K}, \Pi, P] \parallel S'} \ \textit{(ens2)}$$

$$\frac{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\lambda_1} C \qquad S \xrightarrow{\lambda_2} S' \qquad \Pi, \mathcal{I} \vdash \lambda_1 \diamond \lambda_2 \succ \lambda}{\mathcal{I}[\mathcal{K}, \Pi, P] \parallel S \xrightarrow{\lambda} C \parallel S'} \ \textit{(ens3)}$$

a system, by moving all name restrictions at top level, thus putting it into a form to which the first rule can be possibly applied. This manipulation may require the renaming of a restricted name with a freshly chosen one, thus ensuring that the name moved at top level is different both from the restricted names already moved at top level (to avoid name clashes) and from the names occurring free in the other (sub-)systems in parallel (to avoid improper name captures).

On inter-ensemble communication. According to the semantics, two components can interact only if they are part of the same ensemble. Here, we tune the semantics for permitting more complex interaction patterns among two or more components, possibly belonging to different ensembles, by exploiting interaction predicates to regulate them. Therefore, first we extend system labels as follows

$$\lambda \ ::= \ \ldots \ \big| \ \lambda_1 \diamond \lambda_2$$

where label $\lambda_1 \diamond \lambda_2$ denotes the concurrent execution of those transitions corresponding to labels λ_1 and λ_2. Then, we add the rules in Table 5 to the operational rules for systems in Table 3.

Basically, the idea is to generalize the mechanism already present in the operational semantics of systems (see e.g. rule *(enscomm)* regulating intra-ensemble communications), by replacing the authorization predicate $\Pi, \mathcal{I} \vdash \lambda$ with predicate $\Pi, \mathcal{I} \vdash \lambda' \succ \lambda$. The latter, while checking whether a transition can be allowed according to the policy Π in force at $\mathcal{I}$, also translates label λ' into λ.

On ensemble-wide broadcast communication. In [8], we also present an extension of SCEL enabling a sort of multicast communication where the potential

recipients are all the members of the ensembles of which the sender is part of. Communication is anonymous and takes place through the coordinators of these ensembles. Due to lack of space, we refer the reader to [8] for a complete account.

5 How to 'Cook' Your Own SCEL Dialect

In this section, we show how dialects of SCEL can be easily defined by appropriately specifying the parameters of the language. As a concrete example, we demonstrate how KLAIM [9] can be obtained.

In order to define a dialect with specific features, one has to fix the parameters SCEL depends on, that is

1. the language for expressing *policies*, together with an *interaction predicate* and an *authorisation predicate*;
2. the languages for representing knowledge *items* and the *templates* to be used to retrieve these items from the repositories;
3. the language for representing *knowledge* repositories, together with the three *operations*, i.e. withdrawal, retrieval and addition, that we assume provided by each knowledge repository's handling mechanism.

Now, to get KLAIM as a dialect of SCEL, we can make the following choices.

Policies must express KLAIM *allocation environments*. These are functions associating logical names to physical names (i.e. addresses) of the different components, thus regulating components visibility and establishing systems architecture. Therefore, policies are rendered as functions from variables to names. As interaction predicate, we take the *interleaving* one (see [8] for details), which is obtained by interpreting controlled composition as the interleaved parallel composition of the two involved processes, while, as authorisation predicate, we take a predicate that does not block any action.

Knowledge items are sequences of values, i.e. *tuples*, while templates are sequences of values and variables. More generally, a value can result from the evaluation of some given expression e belonging to an appropriate language of EXPRESSIONS. KLAIM in turn is parametric with respect to the language of expressions (we assume that it contains *strings* and *integers*).

Knowledge repositories are multisets of tuples, i.e. *tuple spaces*, providing the three operations of withdrawal, retrieval and addition. The first two use *pattern-matching* wrt a given template to pick a tuple from a tuple space: a tuple matches a template if they have the same number of elements and corresponding elements have matching values or variables; variables match any value of the same type, and two values match only if they are identical. In case more tuples match a given template, one of them is arbitrarily chosen.

In practice, we can complete the syntax of KNOWLEDGE, ITEMS and TEMPLATES as shown in Table 6, where e denotes an EXPRESSION producing values.

If we were interested in capturing alternative versions of KLAIM that, e.g., use types to enforce access control (see, e.g. [10]), we can simply add these types, i.e. functions from names to sets of *capabilities*, to the language of policies.

Table 6. Tuple-based SCEL (e is an EXPRESSION)

KNOWLEDGE:	ITEMS:
$\mathcal{K} ::= \langle t \rangle \;\mid\; \mathcal{K}_1 \parallel \mathcal{K}_2$	$t ::= e \;\mid\; c \;\mid\; P \;\mid\; t_1, t_2$

TEMPLATES:
$T ::= e \;\mid\; c \;\mid\; P \;\mid\; !\,x \;\mid\; !\,X \;\mid\; T_1, T_2$

As for component interfaces, the only meaningful attribute is *id* which is set to the name of the component, while attributes *ensemble* and *membership* are set to true and no other attribute is used. At this point, **get**/**qry**/**put**/**new** correspond to KLAIM's **in**/**read**/**out**/**newloc**, while KLAIM's **eval**, that permits to spawn a new process for execution possibly on a remote component, can be rendered in SCEL by means of an appropriate protocol relying on higher-order communication and on action **exec** (see, e.g. [11]).

6 SCEL at Work

In this section we show how the SCEL dialect defined in Section 5 can be used to model a simple yet illustrative example. In particular, we will mainly focus on the *goal-oriented* interaction among SCs and SCEs that are the novelty of our proposal.

In our application scenario, we consider a collection of service components, *all* offering the same services. Each component manages and elaborates service requests with different policies, roughly summarized by the following three quality levels: *gold*, *silver* and *base*. These policies are defined via a combination of predicates on the hardware configuration and the runtime state. For example, the runtime state can give a measure of the number of service requests currently handled locally. The parameters of the different policies are identified by suitable attributes of the component interfaces. In particular, we assume that the tuples $\langle$"*attr*", "*hw*", $i\rangle$ and $\langle$"*attr*", "*load*", $p\rangle$ are stored in the local tuple space of each component. For example, value i (an integer in $[0, 10]$) in the tuple $\langle$"*attr*", "*hw*", $i\rangle$ gives an indication of the capacity of the hardware configuration of the component; while value p (an integer in $[0, 100]$) in the tuple $\langle$"*attr*", "*load*", $p\rangle$ estimates the actual computational load of the component. Notice that the hardware measure is static while the load estimate is updated whenever a component receives or completes a service request.

Each service component also publishes in its interface the signature of the available services through suitable attributes. Here we assume that *aService* is the name of the available service and requires a *string* as input parameter and yields a *string* value as a result. Furthermore, additional information about the client and the session has to be provided when the service is invoked.

Service components constitute three ensembles depending on the *quality of service* they can provide. In particular, we consider three components, named c_g, c_s and c_b, each of which coordinates the service components operating at *gold*, *silver* and *base* level, respectively. Each of these components acts as a *proxy* for

the replicated services. The ensemble coordinator accepts client invocations and allows service components to retrieve them. Then, the invoked service component sends the obtained results back to the client component.

Since ensemble aggregation is goal-oriented, the following predicates

- $S_g(\mathcal{I}) = \mathcal{I}.hw \geq 7$
- $S_s(\mathcal{I}) = (\mathcal{I}.hw \geq 4) \wedge (S_g(\mathcal{I}) \rightarrow \mathcal{I}.load < 40)$
- $S_b(\mathcal{I}) = (S_s(\mathcal{I}) \rightarrow \mathcal{I}.load < 40) \wedge (S_g(\mathcal{I}) \rightarrow \mathcal{I}.load < 20)$.

are assigned to attribute *ensemble* of the three coordinating components c_g, c_s and c_b, respectively. Thus, the *gold* ensemble identifies a gold component by the high measure of its hardware configuration (value greater or equal to 7). The *silver* ensemble is less demanding: a component has to provide an hardware configuration with a level that is at least 4 and, whenever a component provides a hardware configuration that is valued more than 7, the computational load must be less than 40% ($\rightarrow$ stands for logical implication). This last condition guarantees that gold components can handle requests at silver level only when their computational load is under 40%. The same schema is used to define the *base* ensemble. Of course, all the components, independently of their hardware level, can be part of this ensemble. However, gold and silver components are involved only when their computational load is under 20% and 40% respectively.

Notice that components dynamically and transparently leave or enter an ensemble when their computational load changes. For instance, a *gold* component (i.e. a component with attribute *hd* that is greater or equal to 7) leaves a *silver* ensemble whenever its computational load becomes higher than 40%.

The process running at the client component taking care of the interaction with the service, let us call it c, performs the following code fragment:

$$\mathbf{put}(\text{“}invoke\text{”}, \text{“}aService\text{”}, v, c, s)@u.\mathbf{get}(\text{“}result\text{”}, \text{“}aService\text{”}, !x, c, s)@\mathsf{self}.P$$

The client posts the invocation in the tuple space of the coordinator of the ensemble (u is a variable assuming value among c_b, c_s or c_g). Value v is the required input string, while the pair c, s provides the bookkeeping information: c is the client name and s is a value representing the working session. After issuing the invocation, the client waits for the result (recall that action **get** is blocking). Whenever the result of the service invocation is made available, the client can withdraw it from the local tuple space and continue as process P.

Processes running at service components execute the following code fragment:

$$\begin{array}{l}\mathbf{get}(\text{“}invoke\text{”}, \text{“}aService\text{”}, !Param, !Client, !Session)@u.\\ \mathbf{get}(\text{“}attr\text{”}, \text{“}load\text{”}, !x)@\mathsf{self}.\\ \mathbf{put}(\text{“}attr\text{”}, \text{“}load\text{”}, (x+5))@\mathsf{self}.\\ \mathbf{exec}(Q)\end{array}$$

The process is triggered by a client request retrieved from the coordinator's repository. Whenever this happens, the computational load is updated[1] and the

[1] Here we assume each service instance uses 5% of the component computational resources.

process Q, which actually computes the result of the invoked service "$aService$", is executed. We assume that, before its termination, process Q updates the value of attribute *load* and puts the result of the computation into the tuple space of the requesting client.

The application scenario discussed above exploits different forms of communication. First, the invoking client uses inter-ensemble communication for putting its request in the coordinator's repository. Then, the service component uses standard (intra-ensemble) communication to retrieve the request from the coordinator's repository. The processing of the request increases the computational load of the component, which may cause the service component to leave the ensemble where the service request has been retrieved. Therefore, when the service completes, the result is sent back to the client's repository by using inter-ensemble communication. Afterwards, the result can be retrieved by the client through local communication.

7 Adaptation in SCEL

In this section we argue that adaptation can be naturally expressed in SCEL. As we have seen in Section 3, the knowledge repository of components can contain both application data and awareness data. At this level of abstraction, we are not concerned with the way data are actually represented, we only assume that they can be appropriately tagged to distinguish awareness data from application data. This distinction is indeed crucial, as it is at the basis of a tangible notion of *adaptation* [12], which is defined as the run-time modification of awareness data. A component is then deemed *adaptive* if it has a precisely identified collection of awareness data that are modified at run-time, at least in some of its computations. Besides, it is *self-adaptive* if it is able to modify its own awareness data at run-time.

In general, a component in SCEL is adaptive (and, hence, autonomic) because its awareness data can be dynamically modified by means of the actions **put**/**get**/**qry**. Moreover, a component is self-adaptive as the hosted process can trigger modifications of its awareness data by interacting with the local knowledge handler. So-called *feedback-loops*, that adapt behavior of autonomic components to changing contexts, can thus be easily implemented.

The one outlined above is perhaps the simplest form of adaptation, but we can envisage more sophisticated forms by taking the nature of the awareness data into account. Suppose, for example, that the process part of a component is split into an *autonomic manager* controlling execution of a *managed element*. The autonomic manager monitors the state of the component, as well as the execution context, and identifies relevant changes that may affect the achievement of its goals or the fulfillment of its requirements. It also plans adaptations in order to meet the new functional or non-functional requirements, executes them, and monitors that its goals are achieved, possibly without any interruption[2].

[2] The whole body of activities mentioned above has been named MAPE-K loop (Monitoring, Analyzing, Planning, and Executing, through the use of Knowledge) [2].

In practice, the autonomic manager implements the rules for adaptation. Now, by exploiting SCEL higher-order features, namely the capability to store/retrieve (the code of) processes in/from the knowledge repositories and to dynamically trigger execution of new processes (by means of action **exec**), it is e.g. possible to dynamically replace (part of) the managed element process or even the autonomic manager process. In this case, we are also changing the rules, i.e. processes, with which the awareness data are manipulated, since these rules are represented as awareness data themselves.

A managed element can be seen as an empty "executor" which retrieves from the knowledge repository the process implementing a required functionality *id* and bounds it to a variable X, sends the retrieved process for execution and waits until it terminates (this coordination can be worked out by exchanging appropriate synchronisation items). Also actual parameters for the process to be executed can be stored as knowledge items and retrieved by the executor (or by the process itself) when needed, as shown by the code fragment below.

$$\begin{aligned} ME \triangleq\ & \mathbf{qry}(\text{``}required_functionality_id\text{''}, !X)@\mathsf{self}.\\ & \mathbf{get}(\text{``}required_functionality_id_args\text{''}, !y, !z)@\mathsf{self}.\\ & \mathbf{exec}(X(y,z)).\\ & \mathbf{get}(\text{``}wait_termination_id''))@\mathsf{self}.ME \end{aligned}$$

Items containing processes or parameters can be thought of as awareness data. Autonomic managers can add/remove/replace these data from the knowledge repositories thus implementing the adaptation logic and therefore changing the managed element behavior. For example, different versions of the process providing a requested service may exist. While managed elements could only read these data, the autonomic manager could dynamically change the association between the service request and the service process by simply performing:

$$\begin{aligned} & \mathbf{get}(\text{``}required_functionality_id\text{''}, !X)@\mathsf{self}.\\ & \mathbf{put}(\text{``}required_functionality_id\text{''}, Q)@\mathsf{self}. \end{aligned}$$

which has the effect of replacing the 'old' service implementing the functionality *id* with a possibly new one Q.

The autonomic manager can also add a new service or even remove an existing one. Besides, it is a process just like the managed element, thus it is very well suited to be itself subject to adaptation. In this way we can build up hierarchical adaptations and cover a wide range of adaptation mechanisms.

One issue with SCEL is that it does not have any specific mechanism for stopping or killing processes. However, exploiting knowledge and higher-order features, the application designer can specify when to terminate processes by following suitable patterns. For example, in the code fragment below, the managed element can ask the autonomic manager for the authorization to proceed as process P and, in the negative case, signal its termination.

$$\begin{aligned} & \mathbf{qry}(pid, \text{``}ko\text{''})@\mathsf{self}.\mathbf{put}(pid, \text{``}dead\text{''})@\mathsf{self}.\mathbf{nil}\\ + & \mathbf{qry}(pid, \text{``}ok\text{''})@\mathsf{self}.P \end{aligned}$$

This would allow an autonomic manager to send a termination request to the process with identifier *pid* and wait for its termination, assuming that both items (*pid, "ok"*) and (*pid, "ko"*) are used for coordination purposes.

$$\mathbf{get}(pid, \text{“}ok\text{”})@\mathsf{self}.\mathbf{get}(pid, \text{“}dead\text{”})@\mathsf{self}$$

As we have seen, it is the autonomic manager to choose *which* adaptation to use. The decision about *when* to perform adaptation is jointly taken by the autonomic manager and the application designer. This is reminiscent of another approach, named *context-oriented programming* (COP) [13]. COP is a novel programming paradigm introduced to manage and control adaptivity of programs. It allows developers to define *behavioral variations*, chunks of code that can be activated depending on the current working environment (the context), to dynamically modify program execution and thus adapt to its environment. In this approach, the application designer has to insert *adaptation hooks* in the application code and is thus able to control when adaptation can take place. Leaving the designer to specify where and when to adapt has its advantages, because adaptations would be explicit in the code and thus more visible, and the application designer could better plan some adaptations. However, not being transparent to the application designer has significant disadvantages, because only adaptations planned at design phase could be exploited. When the autonomic computing approach is used, the autonomic manager, which continuously monitors awareness data or event occurrences, reacts to changes of contexts or of goals.

Other than language-level adaptation, as e.g. used in COP, another approach to adaptation focuses on the architectural-level. It consists in dynamically reshaping the structure of the system, e.g. by exchanging a specific component with one that provides similar functionalities, but behaves better in a new context. SCEL supports also this coarse-grained approach since component's membership to ensembles is dynamic. Indeed, the membership attribute of a component's interface can be parametric w.r.t. to some information controlled by an autonomic manager.

Finally, in case of distributed applications one can plan to have (*i*) awareness data residing at autonomic elements and the autonomic managers performing the adaptation for all controlled elements, or (*ii*) all autonomic elements reading from a single knowledge repository that contains both awareness data and global autonomic processes. The distributed approach may cause consistency problems between autonomic elements during the adaptation procedure, because the autonomic managers of different elements may not be synchronized. The centralized approach may lead to efficiency loss and relies too much on the communication between autonomic elements, that can have considerable latencies or be unreliable. However, both approaches may be useful. For example, at ensemble level, adaptation can be partly centralized, controlled by an autonomic manager, and partly distributed in each component. At system level, the distributed approach better supports the dynamic structures and loosely-coupled components.

8 Related Work

The term "ensemble" has been recently introduced in the literature (see, e.g., [3,1,14]) to denote a category of systems characterized by heterogeneous collections of computing resources, huge number of potential interactions, context-awareness, dynamically changing network topologies, and unreliable communications. A mathematical model of ensembles and their composition has been introduced in [15]. Ensembles and their constituent parts are abstractly described as relations on sets of inputs and outputs. The "black-box" view of adaptivity is then formally defined. This leads to a preorder relation on ensembles which captures the the ability of ensembles to satisfy goals or maximize a performance measure in different environments. Differently from this denotational model, we introduce an operational model of ensembles and a formal language that allows the description of ensembles in a compact and formal way. A language for programming ensembles, named Meld, has been proposed in [16,17]. Meld is a declarative language originally designed for programming overlay networks. It allows ensembles to be programmed as a unified whole from a global perspective and then compiled automatically into fully distributed local behaviors. This approach is somehow reminiscent of Declarative Networking [18], a programming methodology that supports the high level specification of network protocols and services, that are then compiled into a dataflow framework and executed. SCEL, instead, is a formal language that could be used as the core of a programming language for ensembles.

The way in which ensembles are characterized in SCEL resembles the way software elements are dynamically grouped into homogenous clusters in [19], where an architecture for the design of component-based distributed self-adaptive systems is outlined. Indeed, both approaches adopt application-specific metrics. Each cluster is headed by a distinguished component, in charge of supervising it and of gathering information from the rest of the system. The supervision mechanism also identifies situations that trigger adaptations.

Context-Oriented Programming (COP) [13] can also be used to write ensemble applications [20]. It exploits *ad hoc* explicit language-level abstractions to express context-dependent behavioral variations and, notably, their run-time activation. So far, most of the efforts in the field of COP have been directed towards the design and implementation of concrete languages. Only a few papers in the literature provide a foundational account of programming languages extended with COP facilities, as e.g. the object-oriented ones of [21,22,23] and the functional one of [24]. All these approaches are however quite different from ours, that instead focusses on distribution and goal-oriented aggregations and supports a highly dynamic notion of adaptation.

Several works have been proposed that use formal techniques to model autonomic computing systems. For example, [25] presents an approach to develop an autonomic service-oriented architecture. This and other examples (e.g., [26,27]), however, focus on the use of formal techniques for specific target applications.

Our work, instead, aims at introducing general techniques to achieve autonomicity rather than at modeling specific autonomic systems. SCEL formal semantics permits to better understand how autonomicity is obtained.

Core languages designed in the area of concurrency theory are natural candidates for the specification of autonomic systems. Many such formalisms aim at modelling dynamically changing network topologies, a feature common to many types of distributed systems and to ensembles. For example, CWS [28] deals with communication aspects that are specific of wireless communications, while the ω-calculus [29] addresses the modeling problems of mobile ad-hoc networks. We want also to mention [30], that uses the Gamma formalism, a computing model inspired by the chemical reaction metaphor, to develop a higher-order language for specifying autonomic systems, and [31], that presents a biochemical calculus expressive enough to represent adaptive systems, together with a formal framework for property checking.

9 Concluding Remarks and Future Directions

We have introduced SCEL, a new language that brings together various programming abstractions that permit directly representing *knowledge*, *behaviors* and *aggregations* according to specific *policies*, and naturally programming interaction, adaptation and self- and context-awareness. Our *language-based* approach enables us to govern the complexity of the issues under consideration by imposing a structure over the variety of computational entities involved. A further advantage is that all programming abstractions are based on solid semantic grounds. This lays the basis for developing logics, tools and methodologies for formal reasoning about system behavior in order to establish qualitative and quantitative properties of both the individual components and the ensembles.

We are currently assessing to which extent SCEL achieves its goals, i.e. modeling the behavior of service components and their ensembles, their interactions, their sensitivity and adaptivity to the environment. As testbeds we will use three case studies from three different application domains: Robotics (collective transport), Cloud-computing (transiently available computers), and e-Mobility (cooperative e-vehicles). This process might require tuning the language features. After this, we plan to implement SCEL, possibly by exploiting the distributed software framework IMC [32].

We also want to develop a methodology that enables components to take decisions about possible alternative behaviors by choosing among the best possibilities while being aware of the consequences. By relying on an abstract description of the evolving environment, each component will be able to verify locally the possibility (or the probability) of guaranteeing the wanted properties or of achieving the wanted goals by analyzing the possible outcome of its interactions with the abstract model. This kind of information will then be used to take decisions about the choices that a component has to face. This abstract description is not fixed but may change according to the interactions of the component with the rest of the system or as a consequence of the changes in the, possibly imprecise, contextual information in which the entity is currently running.

Our proposal combines notions from different research fields. This will permit the cross fertilization of concepts and techniques. For instance, in the long run, we expect that analytical methods typical of the so called *big data* science can be fruitfully adopted to discover aggregation patterns and, consequently, predict behavior of highly complex SCEs. Understanding how aggregations of SCs may evolve is a key issue for developing optimization techniques.

References

1. Hölzl, M., Rauschmayer, A., Wirsing, M.: Software Engineering for Ensembles. In: Wirsing, M., Banâtre, J.-P., Hölzl, M., Rauschmayer, A. (eds.) Software-Intensive Systems. LNCS, vol. 5380, pp. 45–63. Springer, Heidelberg (2008)
2. IBM: An architectural blueprint for autonomic computing. Technical report, 3rd edn. (June 2005)
3. Project InterLink (2007), `http://interlink.ics.forth.gr/central.aspx`
4. Project ASCENS (2010), `http://www.ascens-ist.eu/`
5. Wirsing, M., Hölzl, M., Tribastone, M., Zambonelli, F.: ASCENS: Engineering Autonomic Service-Component Ensembles. In: Beckert, B., de Boer, F., Bonsangue, M., Damiani, F. (eds.) FMCO 2011. LNCS, vol. 7542, pp. 1–24. Springer, Heidelberg (2012)
6. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I & II. Inf. Comput. 100(1), 1–77 (1992)
7. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. 60-61, 17–139 (2004)
8. De Nicola, R., Ferrari, G., Loreti, M., Pugliese, R.: Languages primitives for coordination, resource negotiation, and task description. ASCENS Deliverable D1.1 (September 2011), `http://rap.dsi.unifi.it/scel/`
9. De Nicola, R., Ferrari, G., Pugliese, R.: Klaim: A Kernel Language for Agents Interaction and Mobility. IEEE Trans. Software Eng. 24(5), 315–330 (1998)
10. Gorla, D., Pugliese, R.: Dynamic management of capabilities in a network aware coordination language. J. Log. Algebr. Program. 78(8), 665–689 (2009)
11. De Nicola, R., Gorla, D., Pugliese, R.: On the expressive power of klaim-based calculi. Theor. Comput. Sci. 356(3), 387–421 (2006)
12. Bruni, R., Corradini, A., Gadducci, F., Lluch Lafuente, A., Vandin, A.: A Conceptual Framework for Adaptation. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 240–254. Springer, Heidelberg (2012)
13. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology 7(3), 125–151 (2008)
14. Want, R., Schooler, E., Jelinek, L., Jung, J., Dahle, D., Sengupta, U.: Ensemble computing: Opportunities and challenges. Intel Technology Journal 14(1), 118–141 (2010)
15. Hölzl, M., Wirsing, M.: Towards a System Model for Ensembles. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 241–261. Springer, Heidelberg (2011)
16. Ashley-Rollman, M.P., Goldstein, S.C., Lee, P., Mowry, T.C., Pillai, P.: Meld: A declarative approach to programming ensembles. In: IROS, pp. 2794–2800. IEEE (2007)

17. Ashley-Rollman, M.P., Lee, P., Goldstein, S.C., Pillai, P., Campbell, J.D.: A Language for Large Ensembles of Independently Executing Nodes. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 265–280. Springer, Heidelberg (2009)
18. Loo, B.T., Condie, T., Garofalakis, M., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative networking. Commun. ACM 52(11), 87–95 (2009)
19. Baresi, L., Guinea, S., Tamburrelli, G.: Towards decentralized self-adaptive component-based systems. In: Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2008, pp. 57–64. ACM, New York (2008)
20. Salvaneschi, G., Ghezzi, C., Pradella, M.: Context-oriented programming: A programming paradigm for autonomic systems. CoRR abs/1105.0069 (2011)
21. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. 23(3), 396–450 (2001)
22. Hirschfeld, R., Igarashi, A., Masuhara, H.: ContextFJ: a minimal core calculus for context-oriented programming. In: Proceedings of the 10th International Workshop on Foundations of Aspect-Oriented Languages, FOAL 2011, pp. 19–23. ACM, New York (2011)
23. Clarke, D., Costanza, P., Tanter, E.: How should context-escaping closures proceed? In: Proc. of COP 2009, pp. 1:1–1:6. ACM, New York (2009)
24. Degano, P., Ferrari, G.-L., Galletta, L., Mezzetti, G.: Types for Coordinating Secure Behavioural Variations. In: Sirjani, M. (ed.) COORDINATION 2012. LNCS, vol. 7274, pp. 261–276. Springer, Heidelberg (2012)
25. Bhakti, M.A.C., Azween, A.: Formal modeling of an autonomic service oriented architecture. In: CSIT, vol. 5, pp. 23–29. IACSIT Press (2011)
26. Li, Z., Parashar, M.: Rudder: An agent-based infrastructure for autonomic composition of grid applications. Multiagent and Grid Systems 1(3), 183–195 (2005)
27. Dong, X., Hariri, S., Xue, L., Chen, H., Zhang, M., Pavuluri, S., Rao, S.: Autonomia: an autonomic computing environment. In: IPCCC, pp. 61–68. IEEE (2003)
28. Mezzetti, N., Sangiorgi, D.: Towards a calculus for wireless systems. Electr. Notes Theor. Comput. Sci. 158, 331–353 (2006)
29. Singh, A., Ramakrishnan, C.R., Smolka, S.A.: A process calculus for mobile ad hoc networks. Sci. Comput. Program. 75(6), 440–469 (2010)
30. Banâtre, J.P., Radenac, Y., Fradet, P.: Chemical Specification of Autonomic Systems. In: IASSE, pp. 72–79. ISCA (2004)
31. Andrei, O., Kirchner, H.: A Higher-Order Graph Calculus for Autonomic Computing. In: Lipshteyn, M., Levit, V.E., McConnell, R.M. (eds.) Graph Theory, Computational Intelligence and Thought. LNCS, vol. 5420, pp. 15–26. Springer, Heidelberg (2009)
32. Bettini, L., De Nicola, R., Falassi, D., Lacoste, M., Loreti, M.: A Flexible and Modular Framework for Implementing Infrastructures for Global Computing. In: Kutvonen, L., Alonistioti, N. (eds.) DAIS 2005. LNCS, vol. 3543, pp. 181–193. Springer, Heidelberg (2005)

A Survey on Basic Connectors and Buffers*

Roberto Bruni[1], Hernán Melgratti[2], and Ugo Montanari[1]

[1] Dipartimento di Informatica, Università di Pisa, Italy
[2] Departamento de Computación, Universidad de Buenos Aires - Conicet, Argentina

Abstract. Recent years have witnessed an increasing interest about a rigorous modelling of (different classes of) connectors. Here, the term connector is used to name entities that can regulate the interaction of possibly heterogeneous components. Thus, connectors must take care of exogenous coordination, handling all those aspects that lie outside the scopes of individual components. This has led to the development of different frameworks that are used to specify, design, analyse, compare, prototype and implement connector-based middleware and a rigorous mathematical foundation of connectors is crucial for the analysis of exogenously coordinated systems. In this survey, we overview the main features of some notable theories of connectors, namely the algebra of stateless connectors, the tile model, Reo, BIP, nets with boundaries and the wire calculus. We discuss similarities, differences, mutual embedding and possible enhancements.

1 Introduction

The inherent complexity of modern distributed systems can only be tackled by modular engineering practices and methodologies that enhance the structural and logical blueprint of such systems. This way, it is possible to prove properties of the system either by construction, assembling well-behaving subsystems according to sound patterns, or by decomposition, dividing the systems and the property to be proved in smaller parts that can be analysed separately. *Component-based design* relies on the separation of concerns between coordination and computation. Component-based systems are built from sequential computational entities, the *components*, that should be loosely coupled w.r.t. the concurrent execution environments where they will be deployed. The component interfaces comprise the number, kind and peculiarities of communication ports. The communication media that make possible to interact are called *connectors*. Intuitively, they can be understood as (suitably decorated) channels or links among the ports of the components. Graphically, ports are represented as nodes and connectors as hyperarcs whose tentacles are attached to the ports they control. Several connectors can also be combined together by merging some of the ports their tentacles are attached to. Semantically, each connector imposes

* Research supported by the EU Integrated Project 257414 ASCENS, the Italian MIUR Project IPODS (PRIN 2008), ANPCyT Project BID-PICT-2008-00319, and UBACyT 20020090300122.

B. Beckert et al. (Eds.): FMCO 2011, LNCS 7542, pp. 49–68, 2013.
© Springer-Verlag Berlin Heidelberg 2013

suitable constraints on the allowed communications among the components it links together. For example, a connector may impose handshaking between a sender component and a receiver component (Milner's CCS-like synchronization), or it may require an agreement on the action to be performed next by all components that it connects (Hoare's CSP-like synchronization). A different kind of connector may trigger the broadcasting of a message sent from one component to all the other linked components. The evolution of a network of components and connectors (just *network* for brevity) can be seen as if played in rounds: At each round, the components try to interact through their ports and the connectors allow/disallow some of the interactions selectively. A connector is called *stateless* when the interaction constraints that it imposes over its ports stay the same at each round; it is called *stateful* otherwise. To address composition and modularity of a system, networks are often decorated with (input and output) interfaces: in the simplest case, they consist of ports through which a network can interact. For example, two networks can be composed by merging the ports (i.e. nodes) they have in common. Ports that are not in the interface are typically private to the network and cannot be used to attach additional connectors. The distinction between input and output ports indicates in which direction the data should flow, but feedback is also possible through short-circuit connectors, which redirects some of the emitted output of a network to (some of) its input.

In this paper we survey some formal approaches to the modelling, composition and analysis of connectors, namely Reo [1], BIP [6], nets with boundaries [25], the algebra of stateless connectors [10], the tile model [17], and the wire calculus [24]. Although the approaches we shall consider are quite different in spirit, we will argue that they are different ways to look at the same entity. We briefly illustrate below the analysed frameworks by following the chronological order in which they were proposed. To expose the analogies and differences of the approaches, we shall use as a running example the modelling of compensation-based workflows typical of the area of business process modelling (see Section 2). We present the essential technical machinery underlying the considered theories in dedicated sections (the presentation order has been guided by practical dependencies arising in the descriptions of the different models). Some final considerations are reported in Section 8.

The Algebra of Stateless Connectors and the Tile Model: An algebra consisting of five kinds of basic stateless connectors (plus their duals) has been presented in [10]. The connectors can be composed in series or in parallel. The operational, observational and denotational semantics of connectors are first formalised separately and then shown to coincide. Moreover, a complete normal-form axiomatisation is available for them.

The Tile Model [17,9] offers a flexible and adequate semantic setting for concurrent systems [22,15,13] and also for defining the operational and abstract semantics of suitable classes of connectors, of which the algebra of stateless connectors is a particular instance. Tiles express the reactive behaviour of connectors in terms of $\langle \text{trigger}, \text{effect} \rangle$ pairs of labels. In this

context, the usual notion of equivalence is called *tile bisimilarity*. Tile bisimilarity is a congruence when a simple tile format is met by basic tiles [17].

The Reo Coordination Model: Reo [1] is an exogenous coordination model based on channel-like connectors that mediate the flow of data among components. Notably, a small set of point-to-point primitive connectors is sufficient to express a large variety of interesting constraints over the behaviour of connected components, including various forms of mutual exclusion, synchronization, alternation, and context-dependency. Components and primitive connectors can be composed into larger Reo circuits by disjoint union up-to the merging of shared Reo nodes. The semantics of Reo has been formalized in several ways, see [19] for a recent survey.

The BIP Component Framework: BIP [6] is a component framework for constructing systems by superposing three layers of modelling, called Behaviour, Interaction, and Priority. At the global level, the behaviour of a BIP system can be faithfully represented by a safe Petri net with priorities, whose single transitions are obtained by fusion of component transitions according to the permitted interactions, and priorities are assigned accordingly. An algebraic presentation of BIP connectors with vacuous priorities is given in [7]. One key feature of BIP is the so-called *correctness by construction*, which allows the specification of architecture transformations preserving certain properties of the underlying behaviour. For instance it is possible to provide (sufficient) conditions for compositionality and composability which guarantee deadlock-freedom. The BIP component framework has been implemented in a language and a tool-set. A compositional version of BIP systems is presented in [11].

Nets with Boundaries and the Wire Calculus: Nets with boundaries takes inspiration from the open nets of [5]. The main idea is that nets are extended with input/output interfaces that can be used by transitions to synchronise their firings with the environment. C/E nets with boundaries can be composed in series and in parallel and come equipped with a labelled transition system that fixes their operational and bisimilarity semantics. The wire calculus [24] is a process algebra whose action prefixes come with an input/output arity typing. In [25,12] a dialect of the wire calculus has been used to give an exact characterisation of a special class of (stateful) connectors that can be alternatively expressed in terms of nets with boundaries.

2 Running Example

We will illustrate the different approaches surveyed in this paper by modelling the basic operator used for defining *Long Running Transactions* (LRT), i.e., transactions that may require long periods of time to complete. The implementation of LRT does not use locking (as usual for database transactions), but it relies instead on a weaker notion of atomicity based on compensations [18]. Compensations are activities programmed ad hoc to recover partial executions of transactional processes. Then, a LRT is a group of activities that must be all

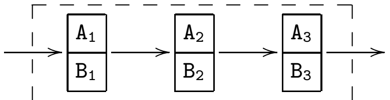
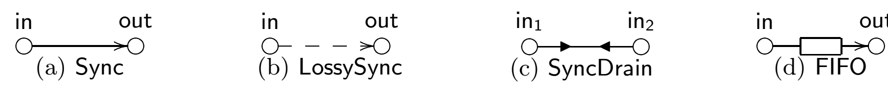


Fig. 1. A sequential saga



(a) Sync (b) LossySync (c) SyncDrain (d) FIFO

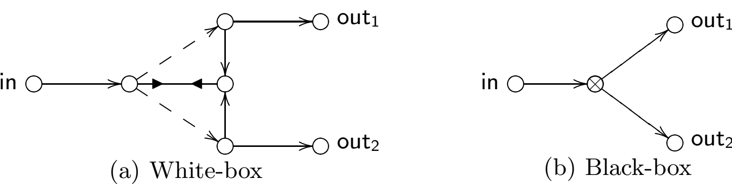
Fig. 2. Graphical representation of Reo basic connectors

either successfully executed or compensated otherwise. Consider the LRT presented in Fig. 1, where the transaction consists in the sequential execution of the activities A_1, A_2 and A_3, that can be compensated respectively by B_1, B_2 and B_3. Suppose now that the activity A_1 completes successfully while the activity A_2 fails. In this case, the failure of A_2 activates the execution of the compensation B_1 to undo as much as possible the effects of A_1, because the transaction failed as a whole. Note that B_2 is not executed, because A_2 has not completed. Differently, if both A_1 and A_2 succeed while A_3 fails, then the compensations will be executed in the reverse order, i.e., first B_2 and then B_1.

Recent years have seen an increasing interest in compensation-based languages for LRT, especially in the area of business process modelling [26], mostly exploiting standard form of composition (sequential, branching, parallel). The *compensation pair* A%B is one key operation common to most of them, whose modelling as connector middleware shall be our running example.

3 The Reo Coordination Model

Reo [1] is a connector-based exogenous coordination model. Connectors are essentially graphs where the edges are user-defined communication channels and the nodes implement a fixed routing policy. Reo *channels* are entities that have exactly two ends, also referred to as *ports*, which can be either source or sink ends: Source ends accept data into, and sink ends dispense data out of their channels. Typical primitive connectors are: (i) the Sync channel, which allows a data item to flow from its source end to its sink end when the latter is able to accept it; (ii) the LossySync channel, similarly to the Sync channel but the data item is lost when the sink end is not ready to accept it; (iii) the SyncDrain channel, which is a channel with two source ends that accept data simultaneously and dispense them subsequently; (iv) the FIFO channel, which is an asynchronous channel with a buffer of capacity one. The set of primitive channels is completed with AsyncDrain, Filter, Transformer, Timer (their definition can be found at [1]). The graphical representation of basic channels is shown in Fig. 2.



(a) White-box (b) Black-box

Fig. 3. Reo exclusive router connector

Components and primitive connectors can be composed into larger Reo circuits by disjoint union, up-to the merging of shared Reo nodes with the same name (this operation is called *join*). Note that a joint node behaves asymmetrically: in input, the node takes non-deterministically a message from one of its incoming channels (the other channels must remain idle); in output, the selected data is written simultaneously to all outgoing channels (that must be able to accept the message). Nodes of a connector can be hidden before composition in order to avoid further joins over those particular nodes. In graphical representation we will leave all hidden nodes unnamed. Figure 3(a) illustrates a well-known composite Reo connector, called *exclusive router*. It joins five Sync, two LossySync and one SyncDrain. The connector provides three visible nodes in, out_1 and out_2. Any data item read on the input port in is written in only one of its output ports out_1 or out_2, depending on which one is ready to consume it. When both out_1 and out_2 are ready to read, then the connector chooses non deterministically one of them. We remark that an input data is never replicated to more than one of its output ports. As a shorthand, we will represent the exclusive router connector as shown in Fig. 3(b).

The semantics of Reo has been formalized in several ways, exploiting, e.g., coalgebraic techniques [3], constraint-automata [4], colouring tables [14], and the tile model [2]. We illustrate here the denotational approach called the two-colour semantics. The two-colour semantics relies on two colours to denote the presence and absence of a message on a port (1 and 0 respectively). The semantics of a connector is defined in terms of the valid assignments of colours to its ports. The tables in Fig. 4 show the valid assignments for some basic connectors. The definition is straightforward for stateless connectors such as Sync, LossySync and SyncDrain. For stateful connectors, constraints have to be provided for any possible state of the connector. Note that the semantics of the FIFO connector is given in terms of two different states, i.e., empty and full. Moreover, the semantic tables define also the state transitions of the connector. Finally, the semantics of a complex Reo circuit corresponds to the set of all possible colour assignments that are consistent with the colouring tables of the involved connectors and reflect the behaviour of joint nodes (i.e., at most one incoming arc has colour 1 and all ongoing arcs have the same colour: 0 when no incoming arc is coloured with 1 and 1 otherwise). In this way, we derive the semantics of complex connectors. For example, valid assignments for $(\mathsf{in}, \mathsf{out}_1, \mathsf{out}_2)$ in the exclusive router are $(0, 0, 0)$, $(1, 0, 1)$ and $(1, 1, 0)$.

Sync

in	out
0	0
1	1

LossySync

in	out
0	0
1	0
1	1

SyncDrain

in	out
0	0
1	1

FIFO

empty			full		
in	out	state	in	out	state
0	0	empty	0	0	full
1	0	full	0	1	empty

Fig. 4. Two-colour semantics of Reo basic connectors

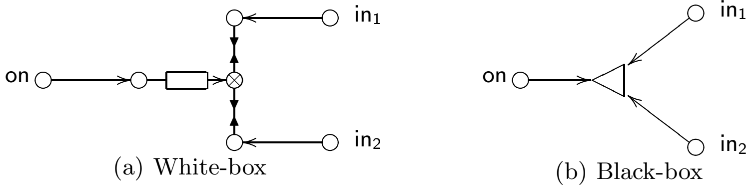


Fig. 5. The stateful selector

3.1 Compensation Pair in Reo

We report here the definition of a compensation pair in Reo as proposed in [21]. We start by presenting an auxiliary connector, named *stateful selector*, which will be used for encoding a compensation pair into a Reo circuit. The stateful selector, depicted in Fig. 5(a), behaves as follows: At the initial state this connector can only accept a message over port on because a synchronization over in_1 and in_2 would require a message in the FIFO channel. A synchronization over on puts a message in the FIFO channel that next enables the synchronization over just one of the ports in_1 and in_2. Note that in_1 and in_2 are in mutual exclusion due to the connector $\otimes$, and hence, the connector can accept just one message on either in_1 or in_2. The connector returns to its initial state after this synchronization. As a shorthand we will depict the stateful selector as shown in Fig. 5(b).

The Reo circuit modelling the compensation pair A%B is shown in Fig. 6. The execution flow starts when a message is written in channel Start, which activates the execution of the activity A. After completion, A will write a message on its output port, which will set the stateful selector and write a message on port Performed for signalling that the activity has been successfully executed (this could serve for instance to activate the next activity in the flow). Eventually, the performed task will be cancelled or committed, after which the effects of a committed task cannot be undone or cancelled anymore. If a Cancel message arrives, the compensation activity B is executed and the task A is considered to be cancelled (this is signalled by sending a message in port Cancelled). If a Commit message arrives, the port Committed emits also a message. The messages to commit or cancel the task are generated from the controller of the transaction. (For simplicity, we omit details here and refer interested readers to [21]).

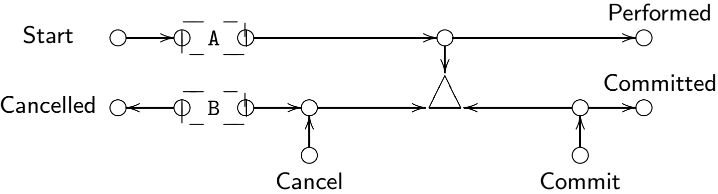


Fig. 6. Reo circuit for the compensation pair `A%B`

4 The BIP Component Framework, and BI(P)

BIP [6] is a component framework that exploits a three-layered architecture: 1) the lower level is called *Behaviour* and it fixes the activities of individual atomic components; 2) the middle layer is called *Interaction* and it defines the hand-shaking mechanisms between components; and 3) the top level is called *Priority* and it assigns a partial order of preferences to the admissible interactions. This section recalls the formal definition of BIP using the notation from [8]. Here we disregard priorities for simplicity, and thus we name BI(P) the presented framework.

The lower layer consists of a set of atomic components with ports. The sets of ports of components are pairwise disjoint, i.e., each port is uniquely assigned to a component. Components are modelled as automata whose transitions are labelled by sets of ports.

Definition 1 (Component). *A* component $B = (Q, P, \rightarrow)$ *is a transition system where* Q *is a set of states,* P *is a set of ports, and* $\rightarrow \subseteq Q \times 2^P \times Q$ *is the set of labelled transitions.*

As usual, we write $q \xrightarrow{a} q'$ to denote the transition $(q, a, q') \in \rightarrow$. We say that a is enabled in q, denoted $q \xrightarrow{a}$, iff there exists q' s.t. $q \xrightarrow{a} q'$. We assume that for all q, q' it holds $q \xrightarrow{\varnothing} q'$ iff $q = q'$.

The second layer consists of connectors that specify the allowed interactions between components.

Definition 2 (Interaction). *Given a set of ports* P*, an* interaction *over* P *is a non-empty subset* $a \subseteq P$*.*

We write an interaction $\{p_1, p_2, \ldots, p_n\}$ as $p_1 p_2 \ldots p_n$ and $a \downarrow_{P_i}$ for the projection of $a \subseteq P$ over the set of ports $P_i \subseteq P$, i.e., $a \downarrow_{P_i} = a \cap P_i$.

Definition 3 (BI(P) system). *A* BI(P) system $B = \gamma(B_1, \ldots, B_n)$ *is the composition of a finite set* $\{B_i\}_{i=1}^n$ *of transitions systems* $B_i = (Q_i, P_i, \rightarrow_i)$ *such that their sets of ports are pairwise disjoint, i.e.,* $P_i \cap P_j = \varnothing$ *for* $i \neq j$*, parametrized by a set* $\gamma \subset 2^P$ *of interactions over the set of ports* $P = \biguplus_{i=1}^n P_i$*. We call* P *the underlying set of ports of* B*, written* $\iota(B)$*.*

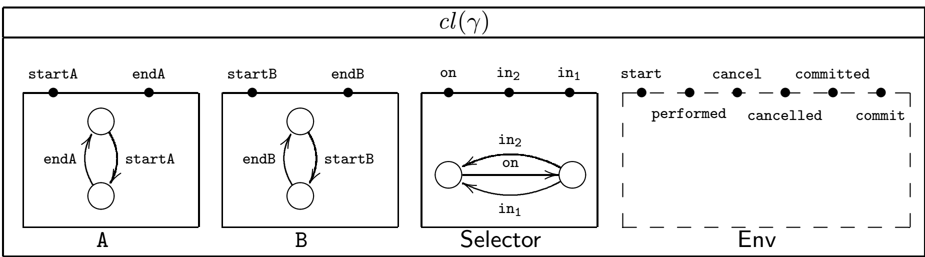


Fig. 7. A simple BIP for compensation pair `A%B`

The semantics of a BI(P) system $\gamma(B_1, \ldots, B_n)$ is given by the transition system $(Q, P, \rightarrow_\gamma)$, with $Q = \Pi_i Q_i$, $P = \biguplus_{i=1}^{n} P_i$ and $\rightarrow_\gamma \subseteq Q \times 2^P \times Q$ is the least set of transitions satisfying the following inference rule

$$\frac{a \in \gamma \qquad \forall i \in 1..n : q_i \xrightarrow{a \downarrow_{P_i}} q_i'}{(q_1, \ldots, q_n) \xrightarrow{a}_\gamma (q_1', \ldots, q_n')}$$

Note that the interactions in γ are pairwise mutually exclusive, e.g., if $a, b \in \gamma$, then it is not necessarily the case that $ab \in \gamma$. We find it convenient to introduce the shorthand $cl(\gamma)$ as the closure of γ w.r.t. set union, i.e., $cl(\gamma)$ is the least set such that $\gamma \in cl(\gamma)$ and $\forall a, a' \in cl(\gamma).\ aa' \in cl(\gamma)$.

4.1 Compensation Pair in BI(P)

A BI(P) system modelling the behaviour of a compensation pair `A%B` is shown in Fig. 7, where

$$\gamma = \{\ \{\texttt{start}, \texttt{startA}\}, \{\texttt{performed}, \texttt{endA}, \texttt{on}\}, \{\texttt{cancel}, \texttt{in}_1, \texttt{startB}\}, \\ \{\texttt{endB}, \texttt{cancelled}\}, \{\texttt{commit}, \texttt{in}_2, \texttt{committed}\}\ \}.$$

As in previous cases, we assume basic activities `A` and `B` to be defined as components with two ports: one for activating its execution (named, `startA` and `startB`, respectively) and other for signalling the completion of the activity (named, `endA` and `endB`, respectively). Moreover, we assume that any initiated execution is completed before starting another execution. Hence, basic activities are modelled as automata with just two states (see Fig. 7). We also rely on a component Selector, which behaves analogously to the stateful selector defined in Reo. In addition, we consider a fourth component Env representing the environment in which the compensation pair will execute. This component acts as the transaction manager that coordinates the execution of the whole transaction. As such, it is in charge of starting the execution of the compensation pair at the proper moment (action `start`) and then it decides whether to `commit` or to

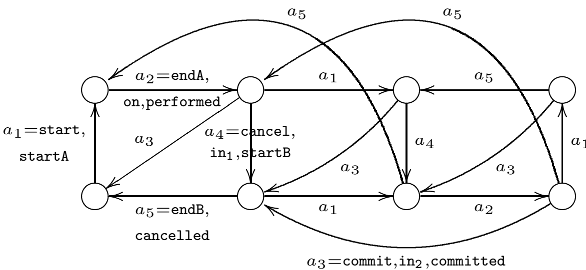


Fig. 8. Synthesized behaviour of the BI(P) system for `A%B`

`cancel` the already executed activity. The synchronization set $cl(\gamma)$ in Fig. 7 defines all the allowed movements of the system. For instance, γ contains just one synchronization for the action `start`, which requires the simultaneous execution of `startA`, i.e., the environment may perform `start` only when the component `A` is able to perform `startA`. Similarly, synchronization $\{\mathtt{endA}, \mathtt{on}, \mathtt{performed}\}$ implies that the termination of `A` enables the component Selector and makes the environment to move with action `performed`. Assuming the more liberal definition of `Env` (that does not introduce deadlocks), the behaviour of the system can be summarized with the LTS in Fig. 8. Note that, if `Env` guarantees that multiple instances of the compensation pair are serialized, then only the four leftmost states are meaningful. Otherwise, it is possible to handle simultaneously up to three instances of the pair: executing one instance of A (third serve), one of B (first serve) with selector on (second serve). We remark that the same behaviour arises in the Reo circuit for the compensation pair when we replace A and B by FIFO connectors.

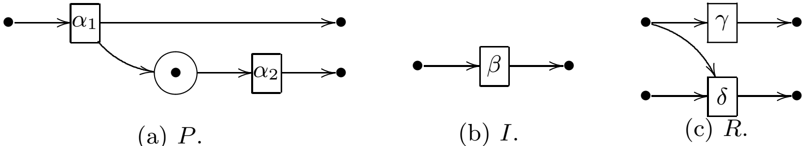
5 Nets with Boundaries

This section summarizes the basic of C/E nets with boundaries introduced in [25]. C/E nets with boundaries are a compositional version of C/E nets that come equipped with a notion of sequential and parallel composition. Contrary to previous proposals in the literature for composing Petri nets, the ports in their boundaries are neither places nor transitions, but rather handshaking points.

We start by describing ordinary Petri nets and then we introduce nets with boundaries. Petri nets [23] consist of *places*, which are repositories of *tokens*, and *transitions* that remove and produce tokens.

Definition 4 (Net). *A* net *N is a 4-tuple $N = (S_N, T_N, {}^\circ{-}_N, {-}^\circ_N)$ where S_N is the (nonempty) set of places,* $\mathtt{a}, \mathtt{a}', \ldots$*, T_N is the set of transitions,* $\mathtt{t}, \mathtt{t}', \ldots$ *(with $S_N \cap T_N = \varnothing$), and the functions ${}^\circ{-}_N, {-}^\circ_N : T_N \to 2^{S_N}$ assign finite sets of places, called respectively source and target, to each transition.*

Transitions t, u are independent when ${}^\circ t \cap {}^\circ u = t^\circ \cap u^\circ = \varnothing$. This notion of independence allows so-called contact situations. Moreover, it also allows consume/produce loops, i.e., a place p can be both in ${}^\circ t$ and t°. A set U of transitions



(a) P. (b) I. (c) R.

Fig. 9. Three nets with boundaries

is mutually independent when, for all $t, u \in U$, if $t \neq u$ then t and u are independent. Given a set of transitions U let ${}^{\circ}U = \cup_{u \in U} {}^{\circ}u$ and $U^{\circ} = \cup_{u \in U} u^{\circ}$.

Definition 5 (Semantics). *Let $N = (P, T, {}^{\circ}-, -^{\circ})$ be a net, $X, Y \subseteq P$ and $t \in T$. Write:*

$$(N, X) \rightarrow_{\{t\}} (N, Y) \stackrel{\mathsf{def}}{=} {}^{\circ}t \subseteq X \wedge t^{\circ} \subseteq Y \wedge X \backslash {}^{\circ}t = Y \backslash t^{\circ}$$

For $U \subseteq T$ a set of mutually independent transitions, write:

$$(N, X) \rightarrow_{U} (N, Y) \stackrel{\mathsf{def}}{=} {}^{\circ}U \subseteq X \wedge U^{\circ} \subseteq Y \wedge X \backslash {}^{\circ}U = Y \backslash U^{\circ}$$

Note that, for any $X \subseteq P$, $(N, X) \rightarrow_{\varnothing} (N, X)$. States of this transition system will be referred to as markings of N.

For the definition of nets with boundaries we let $\underline{k}$, $\underline{l}$, $\underline{m}$, $\underline{n}$ range over finite ordinals: $\underline{n} \stackrel{\mathsf{def}}{=} \{0, 1, \ldots, n-1\}$.

Definition 6 (Nets with boundaries). *Let $m, n \in \mathbb{N}$. A net with boundaries $N : m \rightarrow n$ is a tuple $N = (S, T, {}^{\circ}-, -^{\circ}, {}^{\bullet}-, -^{\bullet})$ where $(S, T, {}^{\circ}-, -^{\circ})$ is a net and functions ${}^{\bullet}- : T \rightarrow 2^{\underline{m}}$ and $-^{\bullet} : T \rightarrow 2^{\underline{n}}$ assign transitions to the left and right boundaries of N, respectively.*

The representation of the left and right boundaries as ordinals is just a notational convenience. In particular, we remark that the left and the right boundaries of a net are always disjoint.

The notion of independence of transitions extends to nets with boundaries in the obvious way: $t, u \in T$ are said to be *independent* when

$${}^{\circ}t \cap {}^{\circ}u = \varnothing \;\wedge\; t^{\circ} \cap u^{\circ} = \varnothing \;\wedge\; {}^{\bullet}t \cap {}^{\bullet}u = \varnothing \;\wedge\; t^{\bullet} \cap u^{\bullet} = \varnothing$$

Example 1. Figure 9 shows three different nets with boundaries. Places are circles and a marking is represented by the presence or absence of tokens; rectangles are transitions and arcs stand for pre and postset relations. The left interface (right interface) is depicted by points situated on the left (respectively, on the right). Figure 9(a) shows the net $P : 1 \rightarrow 2$ containing one place, two transitions and one token. Nets $I : 1 \rightarrow 1$ and $R : 2 \rightarrow 2$ have no places: the former, called *identity*, forwards tokens received on its input port to the output port; the latter has two competing transitions γ and δ for the tokens arriving on the top-positioned input port, and δ requires also a token from the bottom-positioned input port.

Nets with boundaries can be composed in parallel and in series. Given $N : m \to n$ and $M : k \to l$, their tensor product is the net $N \otimes M : m + k \to n + l$ whose sets of places and transitions are the disjoint union of the corresponding sets in N and M, whose maps ${}^\circ-, -^\circ, {}^\bullet-, -^\bullet$ are defined according to the maps in N and M and whose initial marking is $m_{0N} \oplus m_{0M}$. Intuitively, the tensor product corresponds to draw the nets N and M one above the other.

The sequential composition $N; M : m \to k$ of $N : m \to n$ and $M : n \to k$ is slightly more involved and relies on the following notion of synchronization: a pair (U, V) with $U \subseteq T_N$ and $V \subseteq T_M$ mutually independent sets of transitions such that: (1) $U \cup V \neq \varnothing$ and (2) $U^\bullet = {}^\bullet V$.

The set of synchronisations inherits an ordering from the subset relation, i.e. $(U, V) \subseteq (U', V')$ when $U \subseteq U'$ and $V \subseteq V'$. A synchronisation is said to be minimal when it is minimal with respect to this order. Let

$$T_{N;M} \stackrel{\mathsf{def}}{=} \{(U, V) | U \subseteq T_N, V \subseteq T_M, (U, V) \text{ a minimal synchronisation}\}$$

Notice that any transition t in N (respectively t' in M) not connected to the shared boundary n defines a minimal synchronisation $(\{t\}, \varnothing)$ (respectively $(\varnothing, \{t'\})$) in the above sense. The sequential composition of N and M is written $N; M : m \to k$ and defined as $(S_N \uplus S_M, T_{N;M}, {}^\circ-_{N;M}, -^\circ_{N;M}, {}^\bullet-_{N;M}, -^\bullet_{N;M})$, where pre- and post-sets of synchronizations are defined as

- ${}^\circ(U, V)_{N;M} = {}^\circ(U)_N \uplus {}^\circ(V)_M$ and $(U, V)^\circ_{N;M} = (U)^\circ_N \uplus (V)^\circ_M$
- ${}^\bullet(U, V)_{N;M} = {}^\bullet(U)_N$ and $(U, V)^\bullet_{N;M} = (V)^\bullet_M$.

Intuitively, transitions attached to the left or right boundaries can be seen as transition fragments, that can be completed by attaching other complementary fragments to that boundary. When two transition fragments in N share a boundary node, then they are two mutually exclusive options for completing a fragment of M attached to the same boundary node. Thus, the idea is to combine the transitions of N with that of M when they share a common boundary, as if their firings were synchronized. As in general several combinations are possible, only minimal synchronizations are selected.

Example 2. Let P, I and R be the nets in Fig. 9. Then, the net $(P \otimes I); (I \otimes R)$ obtained as the composition of P, R and two copies of I is shown in Fig. 10.

Sometimes we find convenient to write $N = (S, T, {}^\circ-, -^\circ, {}^\bullet-, -^\bullet, X)$ with $X \subseteq S$ for the net $(S, T, {}^\circ-, -^\circ, {}^\bullet-, -^\bullet)$ with initial marking X and extend the sequential and parallel composition to nets with initial marking by taking the union of the initial markings.

For any $k \in \mathbb{N}$, there is a bijection $\ulcorner_\urcorner : 2^{\underline{k}} \to \{0, 1\}^k$ with

$$\ulcorner U \urcorner_i \stackrel{\mathsf{def}}{=} \begin{cases} 1 \text{ if } i \in U \\ 0 \text{ otherwise} \end{cases}$$

Definition 7 (Semantics). *Let $N : n \to n$ be a net and $X, Y \subseteq P_N$. Write:*

$$(N, X) \xrightarrow[\beta]{\alpha} (N, Y) \stackrel{\mathsf{def}}{=} \exists \text{ mutually independent } U \subseteq T_N \text{ s.t}$$
$$(N, X) \to_U (N, Y), \alpha = \ulcorner {}^\bullet U \urcorner, \text{ and } \beta = \ulcorner U^\bullet \urcorner$$

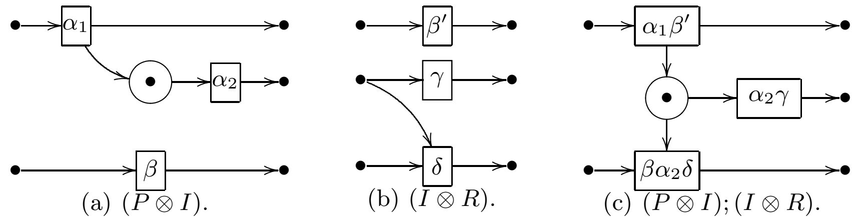


Fig. 10. Composition of nets with boundaries

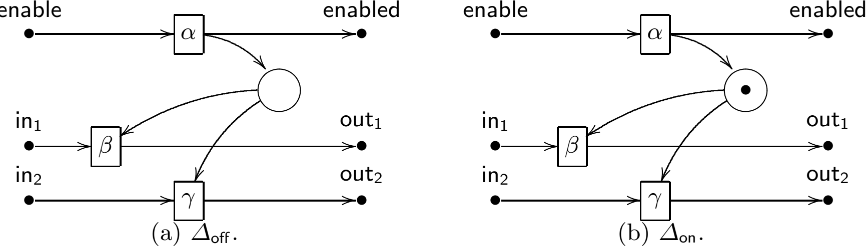


Fig. 11. Net with boundaries for the stateful selector

5.1 Compensation Pair as a Net with Boundaries

We start by introducing an auxiliary net that will be used for encoding a compensation pair into a net with boundaries. Figure 11 depicts a net modelling a stateful selector, analogous to the Reo circuit introduced in Section 3.1. The main difference with the Reo circuit is that we distinguish here between input and output ports (note that nodes in Reo allow both input and output actions). Consequently, we use three input ports enable, in_1 and in_2, which correspond to to the input behaviour of the homonymous nodes in Fig. 5(a). Similarly, the output ports $\mathsf{enabled}$, out_1 and out_2 are considered. Figure 11(a) depicts the initial state of the connector, abbreviated as Δ_{off}, in which selection is not enabled, while Fig. 11(b) shows the state Δ_{on} in which selection is enabled. The allowed movements of the connector are:

1. $\Delta_{\mathsf{off}} \xrightarrow[000]{000} \Delta_{\mathsf{off}}$, i.e., the connector is idle;
2. $\Delta_{\mathsf{off}} \xrightarrow[100]{100} \Delta_{\mathsf{on}}$, i.e., selection has been enabled.
3. $\Delta_{\mathsf{on}} \xrightarrow[000]{000} \Delta_{\mathsf{on}}$, i.e., the connector remains idle;
4. $\Delta_{\mathsf{on}} \xrightarrow[010]{010} \Delta_{\mathsf{off}}$, i.e., input in_1 is chosen;
5. $\Delta_{\mathsf{on}} \xrightarrow[001]{001} \Delta_{\mathsf{off}}$, i.e., input in_2 is chosen.

We remark that Δ_{on} returns to the initial state Δ_{off} after a selection takes place.

We assume any activity A to be modelled as a net with boundaries $[\![A]\!]$ with just one input and one output port, i.e., $[\![A]\!] : 1 \to 1$. In addition, we assume

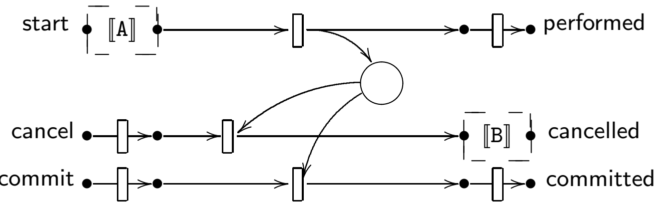


Fig. 12. Net with boundaries for ⟦A%B⟧

that $[\![A]\!]$ is well-defined, and that every started execution ends by signalling the completion of the task with a signal over the output port. We will also use the identity net introduced in Fig. 9(b). Finally, the net corresponding to the compensation pair A%B is the net $[\![\mathtt{A\%B}]\!] : 3 \to 3$ defined as follows

$$[\![\mathtt{A\%B}]\!] = ([\![A]\!] \otimes I \otimes I); \Delta_{\mathsf{off}}; (I \otimes [\![B]\!] \otimes I)$$

A graphical representation of ⟦A%B⟧ is in Fig. 12. Initially, the only allowed movement of the net is the one that initiates the execution of ⟦A⟧ (i.e., a message received on port start). Subsequently, the completion of ⟦A⟧ will be signalled on the output port of the net representing the task. This will fire the top leftmost transition of the net, and consequently a token will be produce in the unique place of the net and a signal will be emitted on port performed. Afterwards, the net will be able to accept one signal on either cancel or commit port. A signal on port cancel will activate the execution of the compensation ⟦B⟧, which will eventually complete and a signal on port cancelled will be produced.

6 Tiles, Wires and the Petri Calculus

The Petri calculus [25] is an algebra of stateful connectors, which basically extends the algebra of stateless connectors from [10] with one-place buffers. It can also be seen as an instance of the tile model or of the wire calculus.

The algebra of stateless connectors [10] consists of five kinds of basic connectors (plus their duals), namely symmetry, synchronization, mutual exclusion, hiding and inaction. The connectors can be composed in series or in parallel. The operational, observational and denotational semantics of connectors are first formalised separately and then shown to coincide. Moreover, a complete normal-form axiomatisation is available for them. These networks are quite expressive: for instance it is shown [10] that they can model all the (stateless) connectors of the architectural design language CommUnity [16]. This result is of particular interest, because it reconciles the algebraic and categorical approaches to system modelling, of which the algebra of stateless connectors and CommUnity are suitable representatives. The algebraic approach models systems as terms in a suitable algebra. Operational and abstract semantics are then usually based on

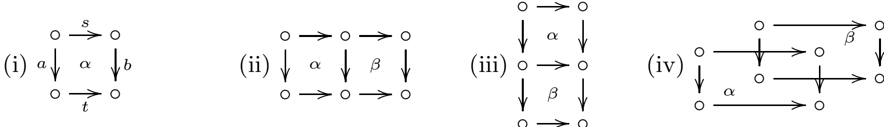


Fig. 13. Examples of tiles and their composition

inductively defined labelled transition systems. The categorical approach models systems as objects in a category, with morphisms defining relations such as subsystem or refinement. Complex software architectures can be modelled as diagrams in the category, with universal constructions, such as colimit, building an object in the same category that behaves as the whole system and that is uniquely determined up to isomorphisms. While in the algebraic approach equivalence classes are usually abstract entities, having a normal form gives a concrete representation that matches a nice feature of the categorical approach, namely that the colimit of a diagram is its best concrete representative.

The tile model [17,9] offers a convenient framework for defining the operational and abstract semantics of connectors. For example, the operational semantics of the algebra of stateless connectors is given in terms of the tile model, which has later been extended to deal with one place buffers in [2]. Also the operational semantics of the Petri calculus, originally defined as a dialect the wire calculus [24], can be straightforwardly represented in the tile model. Tile bisimilarity provided a standard observational congruence in all the above cases.

The name 'tile' is due to the graphical representation of such rules (see Fig. 13). A tile $\alpha : s \xrightarrow[b]{a} t$ is a rewrite rule stating that the *initial configuration* s can evolve to the *final configuration* t via α, producing the *effect* b; but the step is allowed only if the 'arguments' of s can contribute by producing a, which acts as the *trigger* of α (see Fig. 13(i)). Triggers and effects are called *observations* and tile vertices are called *interfaces.*

Roughly, the semantics of component-based systems can be expressed via tiles when: i) components and connectors are equipped with sequential composition $s;t$ (defined when the output interface of s matches the input interface of t), with identities for each interface and with a monoidal tensor product $s \otimes t$ (associative, with unit and distributing over sequential composition); ii) observations have analogous structure $a;b$ and $a \otimes b$. Technically, we rely on configurations and observations that are taken in two monoidal categories that are freely generated from suitable signatures of constructors (i.e. the basic elements of which systems are composed of) and have the same underlying set of objects.

Tiles can be composed horizontally, in parallel, or vertically to generate larger rules. Horizontal composition $\alpha;\beta$ coordinates the evolution of the initial configuration of α with that of β, yielding the 'synchronization' of the two rewrites (see Fig. 13(ii)). Vertical composition is just the sequential composition of

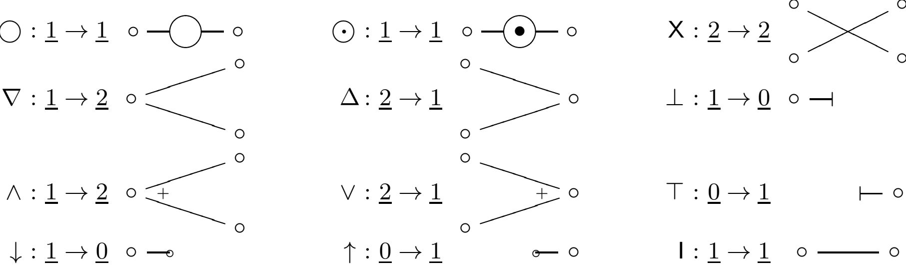


Fig. 14. Graphical representation of Petri calculus constants

computations (see Fig. 13(iii)). The parallel composition builds concurrent steps (see Fig. 13(iv)).

Tiles express the reactive behaviour of connectors in terms of $\langle\text{trigger}, \text{effect}\rangle$ pairs of labels. In this context, the usual notion of bisimilarity over the derived Labelled Transition System is called *tile bisimilarity*. Tile bisimilarity is a congruence (w.r.t. composition in series and parallel) when a simple tile format is met by basic tiles [17].

The wire calculus [24] builds on ideas from [20] to propose a process algebra whose distinctive features are: actions and processes come with an input/output arity typing (that depends on the ports they are using); independent concurrent systems are assembled together using a tensor product $(\cdot \otimes \cdot)$; ports are used instead of channels and communication is possible when ports are linked together by sequential composition $(\cdot ; \cdot)$.

Roughly, we write $\vdash P : (n, m)$ for P with n input ports and m output ports and the usual action prefixes $a.P$ of process algebras are extended in the wire calculus by the simultaneous input of a trigger a and output of an effect b, written $\frac{a}{b}.P$, where a (resp. b) is a string of actions, one for each input port (resp. output port) of the process.

6.1 The Petri Calculus

Terms of the Petri Calculus are defined by the grammar in Fig. 15. It consists of the following constants plus parallel and sequential composition: the empty place $\bigcirc$, the full place $\odot$, the identity wire I, the twist (also swap, or symmetry) X, the duplicator (also sync) ∇ and its dual Δ, the mutex (also choice) $\wedge$ and its dual $\vee$, the hiding (also bang) $\perp$ and its dual $\top$, the inaction $\downarrow$ and its dual $\uparrow$. The graphical representation of Petri calculus constants is in Fig. 14.

Any term has a unique associated *sort* (also called *type*) (k, l) with $k, l \in \mathbb{N}$, that fixes the size k of the left (input) interface and the size l of the right (output) interface of P. The type of constants are as follows: $\bigcirc$, $\odot$, and I have type $(1, 1)$, $\mathsf{X} : (2, 2)$, ∇ and $\wedge$ have type $(1, 2)$ and their duals Δ and $\vee$ have type $(2, 1)$,

$$R ::= \bigcirc \mid \odot \mid \mathsf{I} \mid \mathsf{X} \mid \nabla \mid \Delta \mid \bot \mid \top \mid \wedge \mid \vee \mid \downarrow \mid \uparrow \mid R \otimes R \mid R;R$$

Fig. 15. Petri calculus grammar

$$\frac{R:(k,l) \quad R':(m,n)}{R \otimes R' : (k+m, l+n)} \qquad \frac{R:(k,n) \quad R':(n,l)}{R;R' : (k,l)}$$

Fig. 16. Sort inference rules

$$\frac{}{\bigcirc \xrightarrow[0]{1} \odot} \quad \frac{}{\odot \xrightarrow[1]{0} \bigcirc} \quad \frac{}{\odot \xrightarrow[1]{1} \odot} \quad \frac{}{\mathsf{I} \xrightarrow[1]{1} \mathsf{I}} \quad \frac{}{\nabla \xrightarrow[11]{1} \nabla} \quad \frac{}{\Delta \xrightarrow[1]{11} \Delta} \quad \frac{}{\bot \xrightarrow{1} \bot} \quad \frac{}{\top \xrightarrow[1]{} \top}$$

$$\frac{}{\mathsf{X} \xrightarrow[yx]{xy} \mathsf{X}} \quad \frac{}{\wedge \xrightarrow[x\overline{x}]{1} \wedge} \quad \frac{}{\vee \xrightarrow[1]{x\overline{x}} \vee} \quad \frac{R_1 \xrightarrow[\sigma]{\alpha} R_2 \quad R_1' \xrightarrow[\beta]{\sigma} R_2'}{R_1;R_1' \xrightarrow[\beta]{\alpha} R_2;R_2'} \quad \frac{R_1 \xrightarrow[\beta]{\alpha} R_2 \quad R_1' \xrightarrow[\sigma]{\rho} R_2'}{R_1 \otimes R_1' \xrightarrow[\beta\rho]{\alpha\sigma} R_2 \otimes R_2'} \quad \frac{R:(m,n)}{R \xrightarrow[0^n]{0^m} R}$$

Fig. 17. Operational semantics for the Petri Calculus

$\bot$ and $\downarrow$ have type $(1,0)$ and their duals $\top$ and $\uparrow$ have type $(0,1)$. The sort inference rules for composed processes are in Fig. 16.

The operational semantics is defined by the tiles in Fig. 17, where $x,y \in \{0,1\}$ and we let $\overline{x} = 1 - x$. The labels $\alpha, \beta, \rho, \sigma$ of transitions are binary strings, all transitions are sort-preserving, and if $R \xrightarrow[\beta]{\alpha} R'$ with $R, R' : (n,m)$, then $|\alpha| = n$ and $|\beta| = m$. Notably, bisimilarity induced by such a transition system is a congruence.

Example 3. For example, let $P \stackrel{\mathsf{def}}{=} \nabla;(\mathsf{I}\otimes\odot)$ and $Q \stackrel{\mathsf{def}}{=} \nabla;(\mathsf{I}\otimes\bigcirc)$. It is immediate to check that P and Q have both sort $(1,2)$, in fact we have: $\nabla : (1,2)$, $\mathsf{I}\otimes\odot : (2,2)$ and $\mathsf{I}\otimes\bigcirc : (2,2)$. The only moves for P are $P \xrightarrow[00]{0} P$, $P \xrightarrow[01]{0} Q$ and $P \xrightarrow[11]{1} P$, while the only moves for Q are $Q \xrightarrow[00]{0} Q$ and $Q \xrightarrow[10]{1} P$. It is immediate to note that P is a term analogous to the net in Fig. 9(a).

A close correspondence between nets with boundaries and Petri calculus terms is established in [25], by providing mutual encodings with tight semantics correspondence. First, it is shown that any net $N : m \to n$ with initial marking X can be associated with a term $T_{N,X} : (m,n)$ that preserves and reflects the semantics of N. Conversely, for any term $T : (m,n)$ of the Petri calculus there exists a bisimilar net $N_T : m \to n$. Due to space limitation we omit details here and refer the interested reader to [25].

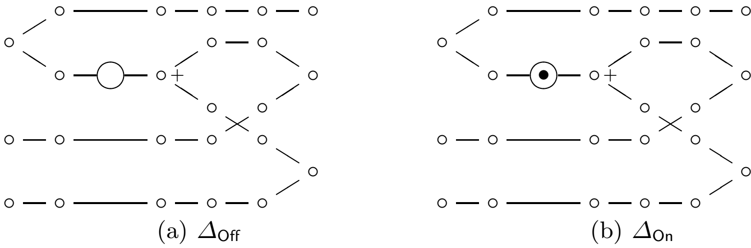


Fig. 18. Petri calculus term for the stateful selector

6.2 Compensation Pair in the Petri Calculus

We start by presenting the Petri calculus term equivalent to the stateful selector, which can be defined as follows:

$$\varDelta_{\texttt{off}} = (\nabla \oplus \mathsf{I} \oplus \mathsf{I}) ; (\mathsf{I} \oplus \bigcirc \oplus \mathsf{I} \oplus \mathsf{I}) ; (\mathsf{I} \oplus \wedge \oplus \mathsf{I} \oplus \mathsf{I}) ; (\mathsf{I} \oplus \mathsf{I} \oplus \mathsf{X} \oplus \mathsf{I}) : (\mathsf{I} \oplus \Delta \oplus \Delta)$$

Its graphical representation is in Fig. 18(a). It can be shown that the reductions of the connector coincide with the movements of the net with boundaries presented in Fig. 11. Moreover, Fig. 18(b) corresponds to the enabled state of the selector. We remark that the Petri calculus term equivalent to the net with boundaries for the stateful selector can be directly obtained by using the encoding defined in [25]. To handle all possible cases, the encoding uses a canonical representation of nets and, as a consequence, obtained terms can become more complex than necessary. For the sake of the simplicity, we prefer to present here a simpler term that is bisimilar to the one produced by the encoding.

Finally, the term representing the compensation pair A%B can be defined analogously to the case of net with boundaries, i.e.,

$$[\![\mathtt{A\%B}]\!] = ([\![A]\!] \otimes \mathsf{I} \otimes \mathsf{I}) ; \varDelta_{\mathsf{off}} ; (\mathsf{I} \otimes [\![B]\!] \otimes \mathsf{I})$$

where $[\![A]\!]$ and $[\![B]\!]$ are Petri calculus terms with sort $(1, 1)$ describing the behaviour of components A and B, respectively.

7 Comparison

BIP and Reo are two prominent approaches for coordination that rely on (apparently) quite unrelated semantic models. In this section we link both models by taking advantage of several results appeared in the literature that formally state correspondences among the approaches presented in the previous sections.

BI(P) and Nets with boundaries. The formal relation between BI(P) and nets with boundaries has been studied in [11]. Firstly, it is shown that any BI(P) system can be mapped into a 1-safe Petri net that preserves computations. Intuitively, the places of the net are in one-to-one correspondence with the states

of the components, while the transitions of the net represent the synchronized execution of the transitions of the components. In addition, [11] introduces a composition operation for BI(P) systems that enables the hierarchical definition of systems in which any BI(P) system can taken as a component of a more complex system. Then, this compositional version of BI(P) systems is used to define a compositional mapping of BI(P) systems into bisimilar nets with boundaries. Finally, it is shown that any net with boundaries without left interface can be encoded as a BI(P) system consisting on just one component. It is in this sense that BI(P) systems and nets with boundaries are retained equivalent.

Nets with boundaries and the Petri Calculus. The technical contribution in [25] enlightens a tight semantics correspondence between these two approaches: it is shown that a Petri calculus process can be defined for each net such that the translation preserves and reflects operational semantics (and thus also bisimilarity). The second result provides the converse translation, from Petri calculus to nets, which requires some technical ingenuity.

Petri Calculus, Wires and Tiles. The wire calculus [24] shares strong similarities with the tile model, in the sense that it has sequential and parallel compositions and exploits trigger-effect pairs labels as observations. However it is presented as a process algebra instead of via monoidal categories and it exploits a different kind of vertical composition. The usual action prefixes $a.P$ of process algebras are extended in the wire calculus by the simultaneous input of a trigger a and output of an effect b, written $\frac{a}{b}.P$, where a (resp. b) is a string of actions, one for each input port (resp. output port) of the process. The Petri calculus is a suitable instance of the wire calculus that roughly models circuit diagrams with one-place buffers and interfaces. An alternative characterization of the Petri calculus as tiles has been given in [12]

Algebra of stateless connectors, Tiles and Wires. The algebra of stateless connectors in [10] can be regarded as a peculiar kind of tile model where all basic tiles have identical initial and final connectors, i.e. they are of the form $s \xrightarrow[b]{a} s$. In terms of the wire calculus, this means that only recursive processes of the form $\mathbf{rec}\, X.\frac{a}{b}.X$ are considered for composing larger networks of connectors.

Tiles and Reo. Differently from the stateless connectors of [10], Reo connectors are stateful (in particular due to the asynchronous one-place buffer connector). Nevertheless, it has been shown in [2] that the two-colour semantics of Reo connectors can be recovered into the setting of the basic algebra of connectors and in the tile approach by adding a connector and a tile for the one-state buffer. It is worth mentioning that, in addition, the tile semantics of Reo connectors provides a description for full computations instead of just single steps (as considered in the original two-colour semantics) and makes evident the evolution of the connector state (particularly, whether buffers get full or become empty).

The main results stating the correspondence among considered approaches are summarized in Fig. 19.

BI(P) —[11]— Nets with boundaries —[25]— Petri Calculus —[11]— Tile —[10,12]— Reo

Fig. 19. Relation among the different models of connectors& buffers

8 Conclusion and Future Work

One of the main limitations of the state-of-the-art theories of connectors is the lack of a reference paradigm for describing and analysing the information flow to be imposed over components for proper coordination. Such a paradigm would allow designers, analysts and programmers to rely on well-founded and standard concepts instead of using all kinds of heterogeneous mechanisms, like semaphores, monitors, message passing primitives, event notification, remote call, etc. Moreover, a reference paradigm would facilitate the comparison and evaluation of otherwise unrelated architectural approaches as well as the development of code libraries for distributed connectors.

Still, some kind of models can be more convenient than others for particular purposes, e.g., if modularity and mantainance is a key issue rather than efficient analysis or automatic synthesis out of requirements. So, we think that having links to move from one model to the other can be as important as having a referential model and, to some extent, it may be more practical.

Some interesting research avenues for future work are (i) the study of suitable extensions of BIP interaction model accounting for dynamically changing topologies of interactions; and (ii) the representation of priorities in approaches such as the algebra of connectors and the tile model. In the former direction, we are studying the possibility to define BIP components whose interfaces can be changed at run time and whose evolution can spawn new component instances and new interaction constraints. This would increase the expressive power of BIP. In the latter direction, we are studying the condition under which global priorities can be safely distributed among connectors. This would allow a modular, inconsistency-free assignment of priorities.

References

1. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science 14(3), 329–366 (2004)
2. Arbab, F., Bruni, R., Clarke, D., Lanese, I., Montanari, U.: Tiles for Reo. In: Corradini, A., Montanari, U. (eds.) WADT 2008. LNCS, vol. 5486, pp. 37–55. Springer, Heidelberg (2009)
3. Arbab, F., Rutten, J.J.M.M.: A Coinductive Calculus of Component Connectors. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) WADT 2002. LNCS, vol. 2755, pp. 34–55. Springer, Heidelberg (2003)
4. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in Reo by constraint automata. Sci. Comput. Program. 61(2), 75–113 (2006)

5. Baldan, P., Corradini, A., Ehrig, H., Heckel, R.: Compositional semantics for open Petri nets based on deterministic processes. Mathematical Structures in Computer Science 15(1), 1–35 (2005)
6. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Fourth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2006, pp. 3–12. IEEE Computer Society (2006)
7. Bliudze, S., Sifakis, J.: The algebra of connectors - structuring interaction in BIP. IEEE Trans. Computers 57(10), 1315–1330 (2008)
8. Bliudze, S., Sifakis, J.: Causal semantics for the algebra of connectors. Formal Methods in System Design 36(2), 167–194 (2010)
9. Bruni, R.: Tile Logic for Synchronized Rewriting of Concurrent Systems. PhD thesis, Computer Science Department, University of Pisa (1999)
10. Bruni, R., Lanese, I., Montanari, U.: A basic algebra of stateless connectors. Theor. Comput. Sci. 366(1-2), 98–120 (2006)
11. Bruni, R., Melgratti, H., Montanari, U.: Connector Algebras, Petri Nets, and BIP. In: Clarke, E., Virbitskaite, I., Voronkov, A. (eds.) PSI 2011. LNCS, vol. 7162, pp. 19–38. Springer, Heidelberg (2012)
12. Bruni, R., Melgratti, H.C., Montanari, U.: A Connector Algebra for P/T Nets Interactions. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 312–326. Springer, Heidelberg (2011)
13. Bruni, R., Montanari, U.: Dynamic connectors for concurrency. Theor. Comput. Sci. 281(1-2), 131–176 (2002)
14. Clarke, D., Costa, D., Arbab, F.: Connector colouring I: Synchronisation and context dependency. Sci. Comput. Program. 66(3), 205–225 (2007)
15. Ferrari, G.L., Montanari, U.: Tile formats for located and mobile systems. Inf. Comput. 156(1-2), 173–235 (2000)
16. Fiadeiro, J.L., Maibaum, T.S.E.: Categorical semantics of parallel program design. Sci. Comput. Program. 28(2-3), 111–138 (1997)
17. Gadducci, F., Montanari, U.: The tile model. In: Proof, Language, and Interaction, pp. 133–166. The MIT Press (2000)
18. Garcia-Molina, H., Salem, K.: Sagas. In: Proceedings of the ACM Special Interest Group on Management of Data Annual Conference, pp. 249–259 (1987)
19. Jongmans, S.-S.T., Arbab, F.: Overview of thirty semantic formalisms for Reo. Scientific Annals of Computer Science 22(1), 201–251 (2012)
20. Katis, P., Sabadini, N., Walters, R.F.C.: Representing Place/Transition Nets in Span(Graph). In: Johnson, M. (ed.) AMAST 1997. LNCS, vol. 1349, pp. 322–336. Springer, Heidelberg (1997)
21. Kokash, N., Arbab, F.: Applying Reo to service coordination in long-running business transactions. In: SAC, pp. 1381–1382 (2009)
22. Montanari, U., Rossi, F.: Graph rewriting, constraint solving and tiles for coordinating distributed systems. Applied Categorical Structures 7(4), 333–370 (1999)
23. Petri, C.: Kommunikation mit Automaten. PhD thesis, Institut für Instrumentelle Mathematik, Bonn (1962)
24. Sobocinski, P.: A non-interleaving process calculus for multi-party synchronisation. In: ICE. EPTCS, vol. 12, pp. 87–98 (2009)
25. Sobociński, P.: Representations of Petri Net Interactions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 554–568. Springer, Heidelberg (2010)
26. Wohed, P., van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M.: Analysis of Web Services Composition Languages: The Case of BPEL4WS. In: Song, I.-Y., Liddle, S.W., Ling, T.-W., Scheuermann, P. (eds.) ER 2003. LNCS, vol. 2813, pp. 200–215. Springer, Heidelberg (2003)

Synthesis-Based Variability Control: Correctness by Construction

Anna-Lena Lamprecht[1], Tiziana Margaria[1],
Ina Schaefer[2], and Bernhard Steffen[3]

[1] Chair for Service and Software Engineering
University of Potsdam, Germany
{lamprecht,margaria}@cs.uni-potsdam.de
[2] Institut für Softwaretechnik und Fahrzeuginformatik
Technische Universität Braunschweig, Germany
i.schaefer@tu-bs.de
[3] Chair for Programming Systems
Technical University Dortmund, Germany
bernhard.steffen@cs.tu-dortmund.de

Abstract. In this paper, we show the power of combining modern synthesis technology with a constraint-oriented approach to variability modeling. This combination guarantees the validity of all the required properties simply by construction: including a new property simply requires adding a corresponding constraint. The synthesis procedure will then automatically take care that all generated variants are property-conform. This fully declarative approach leads to a very agile variability modeling framework, where new product lines guaranteeing new properties can be defined ad hoc and are, due to our synthesis technology, immediately operational. As the underlying constraint language allows fully describing the intended solution space without imposing any over-specification, neither on the structure, nor on the artifacts, our approach may in particular be regarded as a step from the today typical settings with *closed-world assumption* to one with an *open-world assumption*. Impact and ease of this method are illustrated along a small case study running on our prototypical framework implementation.

1 Introduction

Modern software systems exist in many different variants in order to adapt to different customer requirements and application contexts. Software product line engineering [1] aims at developing this family of product variants by managed reuse in order to improve system quality and to decrease time to market. In particular, variability modeling [2–4] is a way of keeping track of the currently supported and used software variants at a high level of abstraction.

Analysis approaches for product lines can be classified in three main directions according to the representation of the analyzed product variants [5]:

1. *Product-based approaches* analyze each product variant in isolation. Therefore, each product variant is generated from the product line representation

B. Beckert et al. (Eds.): FMCO 2011, LNCS 7542, pp. 69–88, 2013.
© Springer-Verlag Berlin Heidelberg 2013

and checked using an existing technique for the considered product variant properties. The advantage of product-based analyses is that the analysis technique does not have to be adapted for product line analysis. The disadvantage, however, is that naive product-based analyses are in general not feasible. The number of possible product variants is exponential in the number of product features, thus, the product-based approach has exponential complexity in the worst case.

A solution to the exponential growth of product variants are subset selection heuristics as applied in product line testing [6]. Here, a subset of the set of possible product variants is determined that provides the covering of the desired feature combinations with the smallest possible set of products. Then, the subset of the products instead of all possible products is tested.

For deductive verification, an incremental product-based analysis approach is suggested in [7]. The incremental verification approach first fully verifies a selected core product of the product line. By analyzing the differences w.r.t. another product variant, it is analyzed which proof obligations are still valid and do not have to be re-established and which proof obligations are affected by the changes and need to re-considered.

2. *Family-based analyses* consider one representation of the complete product line and analyze it in a single pass. Then, it is guaranteed that all products that can ever be generated from the product line have the desired properties. This approach is very popular if the product line is modeled by an annotative approach, since the there the complete product family is already aggregated in one comprehensive structure, the so-called *150%-model* (cf. [8]). For instance, the product line model checking approaches by Classen et al. [9] and Asirelli et al. [10] are based on an annotative product line representation and provide model checking algorithms to efficiently check the family of products in a single model checking run. Also for type systems family-based analyses have been applied, e.g., in FFJ_{PL} [11] and *CFJ* [12]. The advantage of family-based approaches is that the products do not have to be analyzed in isolation. The disadvantages are, however, that family-based analysis approaches rely on a *closed-world assumption* where all possible products have to be known beforehand. If new product variants arise, the complete family analysis has to be re-run. Moreover, the family-based representation easily gets very complex and is hardly scalable for large product lines. In many cases, the family-based representation is also of exponential size in the number of product features, hence no analysis complexity is saved. To overcome these scalability issues, first approaches have been proposed that handle analysis and verification at a more abstract level, for instance by simulation-based model checking [13].
3. *Feature-based analyses* aim at analyzing the reusable building blocks of the product variants in isolation in order to infer the properties of the product variants. Hence, feature-based approaches require a compositional or transformational modeling approach where the reusable product artifacts are modularized. For instance, in [14], a compositional deductive verification technique for delta-oriented software product lines is proposed. It allows

verifying each delta in isolation in order to guarantee that all possible product variants also satisfy their specifications. However, this approach requires a variant of the Liskov principle [15] for the deltas such that their expressiveness is severely restricted. This is in general the case that for feature-based analysis approaches the expressiveness and usage of the product line building blocks has to be restricted in order to achieve a compositional reasoning principle.

Often, combinations of feature-based and product-based or family-based analyses are used in order to avoid the restricted usage of the product line artifacts in solely feature-based analysis. The type system for feature-oriented product lines programmed in LFJ [16] first infers a set of constraints for the feature modules using a feature-based analysis and then generates a large constraint for the complete product line and applies a family-based analysis. The type system for delta-oriented programming [17] takes a similar approach by first applying a feature-based constraint generation phase, but then applying a product-based constraint checking phase.

In any case, combining product line analysis with verification is difficult, since the approaches suffer from tradeoff drawbacks: the richer the family of products, the longer it takes to carry out the analyses and the more restrictive are the requirements to make the approaches work. The most restrictive element seems here to be the model structure.

In this paper, we focus on product line analysis in the context of constraint-based variability modeling [18]. There, the admissible products are described in terms of behavioral (temporal-logic) constraints that ideally exactly capture the necessary frame conditions without imposing any additional design decisions. That is, the variability of the software product line is in fact modeled *completely implicitly*, and no concrete (150%) model structures, which have the tendency to overspecify, are necessary. In particular, this realizes an *open-world assumption*, where new artifacts are seamlessly integrated as soon as they are available. Of course, their proper treatment requires that they are properly integrated into the domain model.

Synthesis techniques are then applied to generate concrete products from these high-level specifications automatically. Hence, analysis and verification of the products become obsolete and are avoided, because the generated products conform to the constraints by design and are thus *correct by construction*. In essence, our approach to correctness by design can be regarded as step from 'construct and verify' to 'constrain and synthesize'.

In fact, this approach comprises the approach to model checking of software product lines of [9, 19]. However, rather than verifying the variability model against some property, we proceed in the following way: In order to verify if a property (or a combination of properties) X is fulfilled by the products that are defined by the variability model, we simply demand for the generation of a product that conforms to $X + M$ (where M are the properties defined by the variability model). If a corresponding product can be generated, X holds. If no product can be generated, X contradicts the variability model. This approach has the benefit that it:

– allows one to avoid overspecification, which easily arises when one is forced to provide concrete (150%) model structures, and
– automatically exploits new artifacts of the domain, as soon as they have been integrated. This is particularly important in domains as agile as the bio-domain discussed in this paper, as there, new artifact arise all the time, and it is impossible for the user (workflow designer) to keep track of all changes of the artifact library. Our synthesis-based approach confronts him gradually with this new functionality, which he can then further constrain to precisely fit his purpose.

The synthesis method underlying our approach (and which has been proposed already almost two decades ago [20–22]) has been successfully applied to a variety of different application scenarios [23–26]. The most comprehensive applications have been developed for the management of variant-rich bioinformatics analysis processes [27–29] based on its recent implementation in the scope of the loose programming paradigm [30] and the PROPHETS framework [31]. They showed that the constraint-driven design and synthesis of variant-rich workflows has the power to enable users to effectively create and manage software processes in their specific domain language. The correctness-by-construction paradigm gradually evolved also from the experiences with student projects as well as from cooperations with academic partners (such as [32] and [33, Chapter 6]) in this domain. It frees non-IT users from dealing with the technicalities of the individual services and their composition, which is the intended goal of our work (see, e.g., [34–37, 27]).

The remainder of this paper is structured as follows. Section 2 introduces our constraint-based variability modeling framework, before Section 3 details on the constraint-driven synthesis of property-conform products based on the variability model. Then, Section 4, illustrates our approach by means of a small case study running on our prototypical framework implementation. Section 5 concludes the paper with a summary, discussion and directives for future work.

2 Constraint-Based Variability Modeling

Constraint-based variability modeling [18, 38] is a liberal variability modeling approach that has a semantic and decision-management touch that is in good alignment with the eXtreme Model Driven Development (XMDD) paradigm of [37, 39]. Instead of defining the set of possible system variants in a bottom-up fashion like most variability modeling approaches, constraint-based variability modeling takes a top-down approach. Based on the set of available reusable components, behavioral constraints define how the components may be combined to form valid system variants. Furthermore, all system variants satisfying the constraints can then be automatically assembled from the components using an automated synthesis algorithm.

Conceptually, constraint-based variability modeling builds on the following philosophy: start with the set of all possible artifact combinations that are compatible with respect to implicit syntactic constraints, and successively restrict

the set of valid artifact combinations by adding behavioral constraints until the desired variability space is reached. This works quite effectively even in the context of quite heterogenous specifications [40–42]. Indeed, the list of constraints is easily extensible: including a new constraint in the verification only requires to write a modal or temporal formula expressing the property to be enforced [43]. Thus, it is possible to describe the intended range of products in a very flexible fashion.

Our prototypical framework implementation builds upon the jABC [44] framework for service-oriented modeling, design, and development of systems. In particular, it makes use of the PROPHETS plugin [31][1], which supports working with processes and workflows by combining semantic annotations, model checking, and automatic synthesis of workflows according to the *loose programming* paradigm [30]. As such, the jABC supports constraint-based variability modeling for families of workflows built as a composition of different artifacts. And what is more, not only a variability model for a software product line can be set up, but, as detailed in Section 3, it is furthermore possible to derive the corresponding products automatically from the specifications.

The following describes the three major ingredients of a PROPHETS-based variability model, namely the domain vocabulary, artifacts characterizations and domain-specific constraints, in greater detail.

2.1 Domain Vocabulary

In constraint-based variability modeling, the properties and characteristics of the available artifacts are described in terms of some suitable domain vocabulary. Within the PROPHETS framework, taxonomies are used to define semantic classifications of types and artifacts that allow for the hierarchical structuring of the domain model. Taxonomies are simple ontologies that relate entities in terms of *is-a* relations. The originally defined artifacts and their input/output types are named *concrete*, whereas their semantic classifications are named *abstract*. This is analogue to the distinction between T-box (abstract concepts) and A-box (concrete, grounded concepts) found in Description Logic [45]. Technically, the artifact and type taxonomies are stored in OWL format [46], and PROPHETS makes use of the OntED plugin [47] to enable their intuitive graphical modeling directly within the jABC framework. In OWL, the concept *Thing* denotes the most general type or artifact and the OWL classes represent abstract classifications. The concrete types and artifacts of the domain are then represented as individuals that are related to one or more of those classifications by *instance-of* relations. Examples of taxonomies are shown in Figures 1 and 2.

2.2 Artifact Characterizations

Constraint-based variability modeling relies on behavioral artifact interface descriptions, whereby artifacts are regarded as transformations that perform particular actions on the available data. Technically speaking, the set of types that

[1] Project homepage: `http://ls5-www.cs.tu-dortmund.de/projects/prophets/`

is available in the domain forms the static aspects (i.e., the type constraints that are used as atomic propositions by the underlying logic), while the set of artifacts represents the dynamic aspects of the domain (which can be used as actions within the specification formulae). In order to enable thorough abstraction from the concrete artifact implementations, artifacts and types are represented by symbolic names throughout the framework.

Each artifact interface is characterized by means of different subsets of the set of all symbolic type names expressed as the USE, GEN, and KILL sets[2]:

USE are the types that must be available before execution of the artifact (i.e. the input types of the artifact), analogously to the *requires* in service-oriented terminology,

GEN is the set of types that are created by the execution of the artifact (i.e. the output types of the artifact), analogously to the *provides* in service-oriented terminology,

KILL defines those types that are destroyed and therefore removed from the set of types that were available prior to execution of the artifact. This has no correspondent in the service-oriented terminology, and it is one of the reasons why we prefer the chosen terminology.

Technically, this artifact meta-information is stored in the form of a simply structured XML file, and is thus clearly decoupled from the concrete artifact implementations. Thus, there is no restriction to the artifacts of a particular platform, and any kind of available artifact can be used in the framework.

2.3 Domain Constraints

The domain-specific constraints must be formalized appropriately, that is, expressed in the Semantic Linear Time Logic, SLTL [20], which is the temporal logic underlying PROPHETS' synthesis method. SLTL is a semantically enriched version of the commonly known propositional linear-time logic (PLTL) that is focused on finite paths consisting of states (here representing system states that are defined by the set of available data types) and transitions (here: artifacts). The syntax of SLTL is defined by the following BNF:

$$\phi ::= true \mid t_c \mid \neg\phi \mid \phi \wedge \phi \mid \langle s_c \rangle \phi \mid G\phi \mid \phi U \phi$$

where t_c and s_c express type and artifact constraints.

Thus, SLTL combines static, dynamic, and temporal constraints. The static constraints are the taxonomic expressions (boolean connectives) over the instances or classes of the type taxonomy. Analogously, the dynamic constraints are the taxonomic expressions over the instances or classes of the artifact taxonomy. The temporal constraints are covered by the modal structure of the logic, suitable to express ordering constraints:

[2] This terminology stems from the standard terminology of data-flow analysis, which is used to analyze which variable values or expression results are available, produced, or invalidated (by rendering them obsolete) at program points of a control flow graph.

- $\langle s_c \rangle \phi$ states that ϕ must hold in the successor state and that it must be reachable with artifact constraint s_c.
- G expresses that ϕ must hold generally.
- U specifies that ϕ_1 has to be valid until ϕ_2 finally holds.

In addition to the operators defined above, it is convenient to derive further ones from these basic constructs, such as the common boolean operators (disjunction, implication, etc.), the eventually operator $\text{F}\phi =_{def} true \text{ U } \phi$, or the weak until operator $\phi \text{ WU } \psi =_{def} (\phi \text{ U } \psi) \vee \text{G}(\phi)$. Furthermore, the next operator $\text{X } \phi$ is often used as an abbreviation of $\langle true \rangle \phi$. A complete formal definition of the semantics of SLTL can be found, for instance, in [22, 30].

Constraints expressed in SLTL comprise two dimensions:

- The *horizontal* dimension, in terms of modalities that describe aspects of relative time, addresses the workflow model and deals with the actual artifact sequences. Horizontal constraints typically constrain the temporal and causal relationship between components and variation points and are given in terms of the temporal logics portion of SLTL.
- Orthogonally, the *vertical* dimension evaluates taxonomies over types and artifacts, allowing for the usage of abstract type and artifact descriptions within the specifications. Vertical constraints specify which components can be instantiated at a certain variation point, making use of the semantics-awareness of SLTL.

Combining the two dimensions allows one to express complex specifications and enables a very flexible fine-tuning of valid variations. In particular, the intent of a workflow can be comfortably specified in terms of temporal logic formulae without unnecessarily constraining the concrete realization. As understanding and writing SLTL constraints can be extremely difficult for users without background in formal methods and inconvenient also for those who are familiar with temporal logics, PROPHETS supports constraint formulation by means natural-language templates (cf., e.g., [31] for details).

3 Constraint-Driven Synthesis

For the automatic generation of products conforming to the variability model, we apply the synthesis algorithm [20] that is included in PROPHETS. It takes two aspects into account: On the one hand, the product must be a valid execution regarding type consistency; on the other hand, the specified constraints must be met. Basically, the algorithm works on a *synthesis universe* and an SLTL formula.

The synthesis universe constitutes the search space in which the synthesis algorithm looks for solutions to the synthesis problem. It combines the artifact descriptions from the domain model into an abstract representation of all possible solutions, and thus contains all artifact sequences that are valid executions, without taking into account any problem-specific information. The synthesis universe is in essence an automaton that connects states with edges according to

available artifacts. While each state represents a subset of all types (abstract and concrete), the connecting edges perform the transition on those types, according to input and output specifications of artifacts, as defined in the domain model. Every path in this automaton, starting from a state that represents the initially available data, constitutes an executable artifact sequence.

The synthesis algorithm interprets the (conjunction of a set of) SLTL formula(e) that express the synthesis problem over paths of the synthesis universe, that is, it searches the synthesis universe for paths that satisfy the given formula(e). The algorithm is based on a parallel evaluation of the synthesis universe and the formula(e). It automatically generates all artifact compositions that satisfy the given specification (i.e., universe and formula(e)), thereby taking the horizontal and vertical dimensions of the constraints into account. Finally, the result of the synthesis algorithm is the set of all specified, executable products variants.

4 Running Example

To illustrate our approach, we present a realization of the frequently applied coffee machine example (cf., e.g., [48, 49]) with the constraint-based framework described in the previous sections. Therefore, Section 4.1 describes the example, before Sections 4.2 and 4.3 show how we realize the constraint-based modeling and constraint-driven synthesis in our framework by presenting the constraint-based variability model and demonstrating the automatic synthesis of constraint-conform coffee machines, respectively. Finally, Section 4.4 compares our method with other approaches that have been applied for realizing this example.

4.1 The Coffee Machine Example

The coffee machine example is woven around a small family of beverage vending machines, which comprises two distinct families of products. The characteristics of this coffee machine product line are described by the following static and temporal ordering constraints (cf., e.g., [48, 49]):

C1. European products exclusively accept one euro coins, while Canadian products exclusively accept one dollar coins.
C2. A customer has to insert a coin first, then choose whether he wants sugar or not, and then select a beverage.
C3. All coffee machines offer coffee, some also tea. Only European products offer cappuccino.
C4. A tone may be rung after delivering the beverage. A ring tone must be rung for cappuccino.
C5. After the cup has been taken by the customer, the machine returns to idle state.

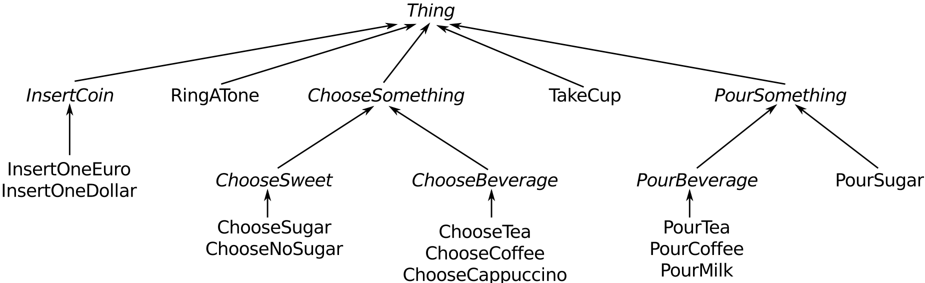


Fig. 1. Artifact taxonomy for the coffee machine product family

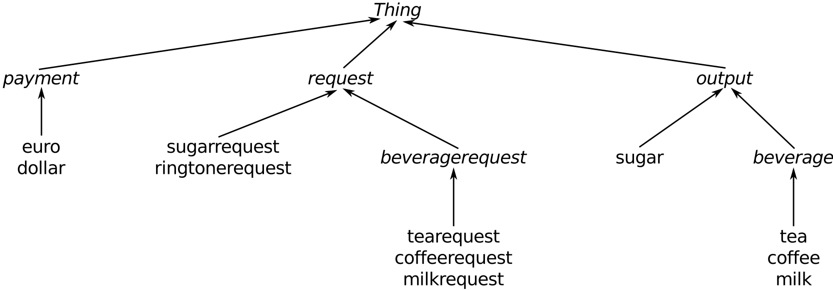


Fig. 2. Type taxonomy for the coffee machine product family

4.2 Constraint-Based Modeling of the Coffee Machine Family

As described in Section 2, a constraint-based variability model consists of a domain-specific vocabulary in terms of artifact and type taxonomies, artifact interface descriptions in terms of the vocabulary, and temporal-logic constraints that specify additional properties of the intended products.

Figures 1 and 2 show the artifact and type taxonomies, respectively, that have been defined for the coffee machine domain model. The artifact taxonomy groups the artifacts of the coffee machine product family (which correspond to the individual actions that can be performed) according to abstract, semantic categories. For instance, the artifacts `InsertOneEuro` and `InsertOneDollar` are both classified as *InsertCoin* artifacts, whereas `RingATone` is not put into a particular category. The abstract class *ChooseSomething* is divided further into *ChooseSweet* and *ChooseBeverage*, into which the corresponding artifacts for opting for sugar/no sugar and for choosing tea/coffee/cappuccino are sorted, respectively. Similarly, the type taxonomy defines a hierarchical ordering for all involved types, which are in this case *payments* (like `euro` and `dollar`), a number of different requests (for sugar, beverages or ringtones), and the actually delivered `sugar` and *beverages* themselves.

The terminology defined by the type taxonomy is then used to describe the interfaces of the artifacts in terms of their individual behavioral constraints.

Table 1. Artifacts in the coffee machine domain model

Artifact	**Use**	**Gen**	**Kill**
InsertOneEuro		euro	
InsertOneDollar		dollar	
ChooseSugar	*payment*	sugarrequest	
ChooseNoSugar	*payment*		
ChooseCoffee	*payment*	coffeerequest	*payment*
ChooseTea	*payment*	tearequest	*payment*
ChooseCappuccino	euro	coffeerequest, milkrequest, ringtonerequest	*payment*
PourSugar	sugarrequest	sugar	sugarrequest
PourMilk	milkrequest	milk	milkrequest
PourCoffee	coffeerequest	coffee	coffeerequest
PourTea	tearequest	tea	tearequest
RingtATone	*beverage*		ringtonerequest
TakeCup	*beverage*		*beverage*

Table 1 lists the artifacts of the coffee machine example along with their interface descriptions in terms of their Use, Gen and Kill sets (cf. Section 2).

The *InsertCoin* artifacts define no input constraints and simply generate the respective *payment*. The different *ChooseSomething* artifacts require a *payment* to be available, which they however kill, and generate the corresponding requests. Addressing the requirements implied by characterizations C3 (only European products offer cappuccino) and C4 (a ringtone must be rung for cappuccino), the `ChooseCappucino` artifact requires a `euro` to be available and generates an additional request for a ringtone. The *PourSomething* artifacts work off the respective requests and deliver (generate) the requested beverage or sugar. The `RingATone` artifact can only be applied when a *beverage* has been delivered (C4). It can also be applied without an explicit `ringtonerequest` from the `ChooseCappuccino` artifact (C4), but it will kill such a request if it is available. The `TakeCup` artifact can be used when a *beverage* is available, which will naturally be not be available any more (killed) when the cup has been taken.

In order to cover those of the characteristics C1–C5 that are not yet covered by the individual artifact descriptions, we have defined the following domain constraints for the coffee machine product line:

– Using the coffee machine begins with inserting a coin, then opting for sugar or not, and then selecting a beverage (C2):

$$\langle InsertCoin\rangle\langle ChooseSweet\rangle\langle ChooseBeverage\rangle true$$

– The last step (before the machine returns to idle) is to take the cup (C5):

$$(F\langle TakeCup\rangle true \& G(\langle TakeCup\rangle true \Rightarrow \neg XX true))$$

– A tone may be rung when beverage delivery has finished (part of C4):

$$G(\langle RingATone\rangle true \Rightarrow XG(\neg\langle PourSomething\rangle true)\wedge\neg\langle RingATone\rangle true)$$

Furthermore, we added the following domain constraints to express some additional essential properties:

– Payments must not be allowed more than once (per iteration):

$$G(\langle InsertCoin\rangle true \Rightarrow XG\neg\langle InsertCoin\rangle true)$$

– All requests must be served appropriately:

$$G(sugarrequest \Rightarrow F(\langle PourSugar\rangle true))$$

$$G(tearequest \Rightarrow F(\langle PourTea\rangle true))$$

$$G(coffeerequest \Rightarrow F(\langle PourCoffee\rangle true))$$

$$G(milkrequest \Rightarrow F(\langle PourMilk\rangle true))$$

$$G(ringtonerequest \Rightarrow F(\langle RingATone\rangle true))$$

The requirements implied by the characteristics C1 and C3 are product-specific constraint, that is, constraints that describe properties of particular products of the coffee machine family. We do not include this kind of constraints in the general variability model, because they do not belong to the description of the potential, but rather to the selection of specific subsets that constitute a product. Correspondingly, we use them for synthesizing the corresponding products, as detailed in the next section.

4.3 Synthesis of Constraint-Conform Coffee Machines

When we apply the synthesis algorithm to the variability model described in the previous section without any further constraints, the generated solutions constitute a *150% model* of the coffee machine product family. Typically, our synthesis framework identifies all linear workflows that are admissible given a particular variability model, from which the user then selects one solution. Figure 3 shows the list of all sequences of action that are possible for the 150% coffee machine model. The framework is, however, also able to combine all linear solutions into an automaton representation, which then corresponds to the actually intended coffee machine models.

Note that the variability model does not constrain the order in which the sugar and the different liquids are to be delivered. Thus, as also visible from Figure 3, some coffee machine traces are simply permutations of others. Conveniently, the PROPHETS synthesis plugin can be configured to automatically remove semantically equivalent permutations from the solutions, so that less results remain to visualize and the corresponding automata become smaller and easier to inspect. For better readability, only accordingly filtered results are shown in

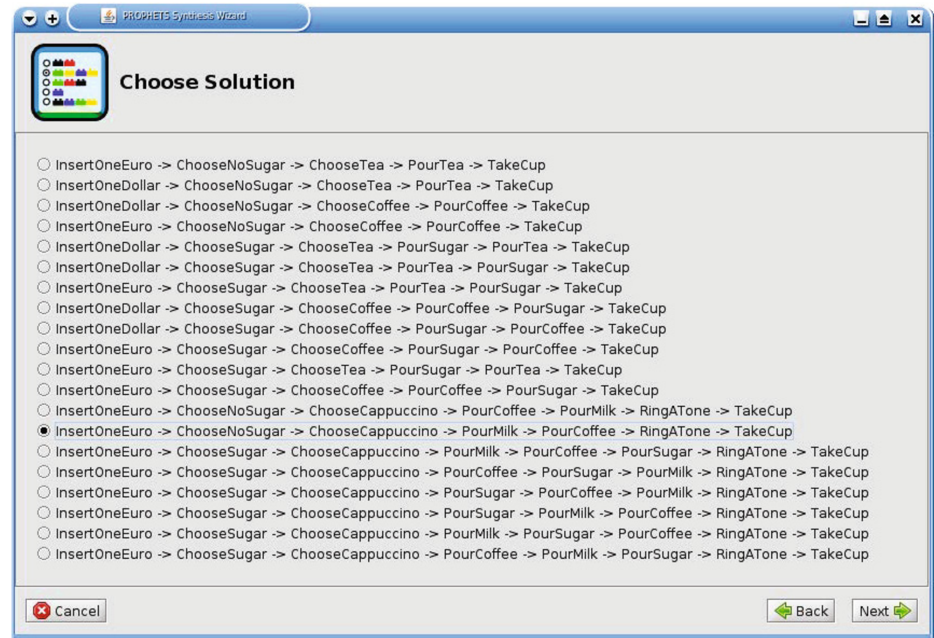


Fig. 3. Coffee machine traces

the following. Figure 4 shows the automaton corresponding to the 150% coffee machine model as described above, but freed from permutations.

As defined by characteristics C1 and C4, the difference between European and Canadian coffee machines is on the one hand that the former accepts only Euros and the latter only Dollars, and on the other hand that only the European machine offers cappuccino. Since the `ChooseCappuccino` artifact has already been specified as only admissible when a Euro has been paid, and as one of the domain constraints expresses that only one coin can be inserted in one iteration, it is sufficient to enforce the use of either `InsertOneDollar` or `InsertOneEuro` in order to obtain the Canadian/European version:

$$F(\langle InsertOneDollar\rangle true)$$

$$F(\langle InsertOneEuro\rangle true)$$

According to characteristic C3, tea does not have to be offered by the coffee machines. To create a coffee machine that does not offer tea, we can simply add an additional constraint that excludes the `ChooseTea` artifact from the solution:

$$G(\neg\langle ChooseTea\rangle true)$$

Figures 5 and 6 show the results obtained for these constraint combinations. Exactly as defined by the constraints, the Canadian machines shown in Figure 5

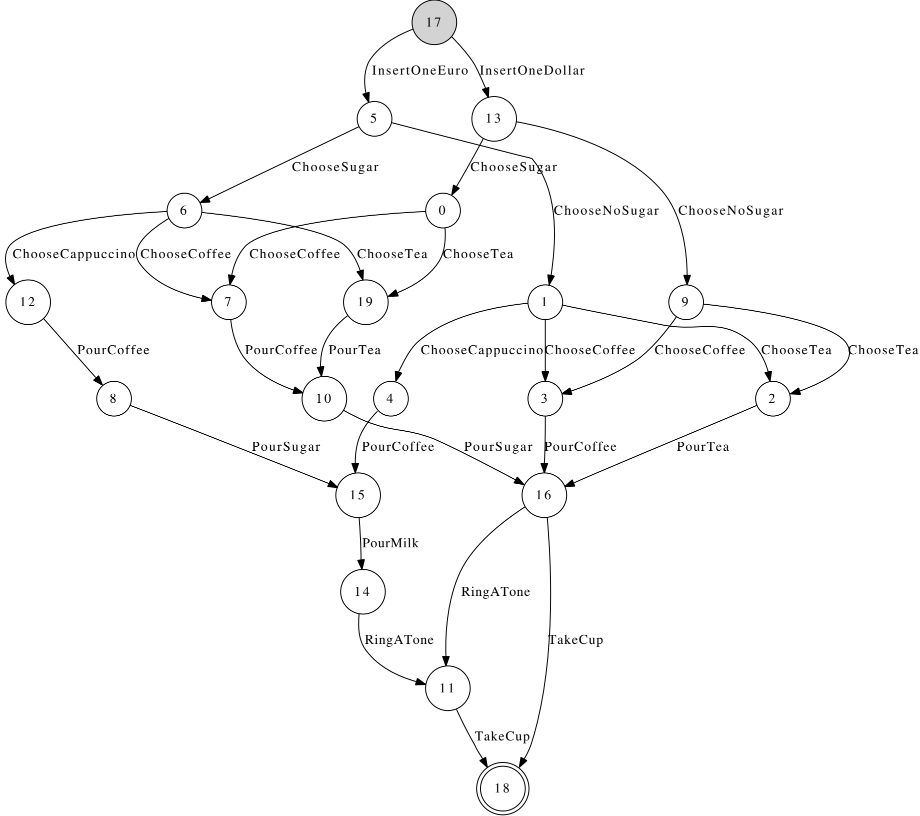


Fig. 4. Synthesized 150% model of the coffee machine

accept exclusively one dollar coins and do not not offer cappuccino. The left variant also offers tea, whereas `ChooseTea` has been excluded from the right one. Analogously, the European coffee machines depicted in Figure 6 accept exclusively one euro coins and offer cappuccino. Furthermore, a tone is definitely rung whenever cappuccino is delivered, whereas the delivery of other beverages may be accompanied by a ringtone or not.

If problem-specific constraints are specified that contradict the variability model, the synthesis problem becomes over-constrained and the algorithm simply returns no solutions. This happens, for instance, when trying to enforce a coffee machine that accepts dollars and offers cappuccino, or a machine that accepts both dollars and euros.

4.4 Comparison with Other Approaches

The coffee machine example has already been used for illustrating and comparing two other, related approaches to software product line engineering [49]:

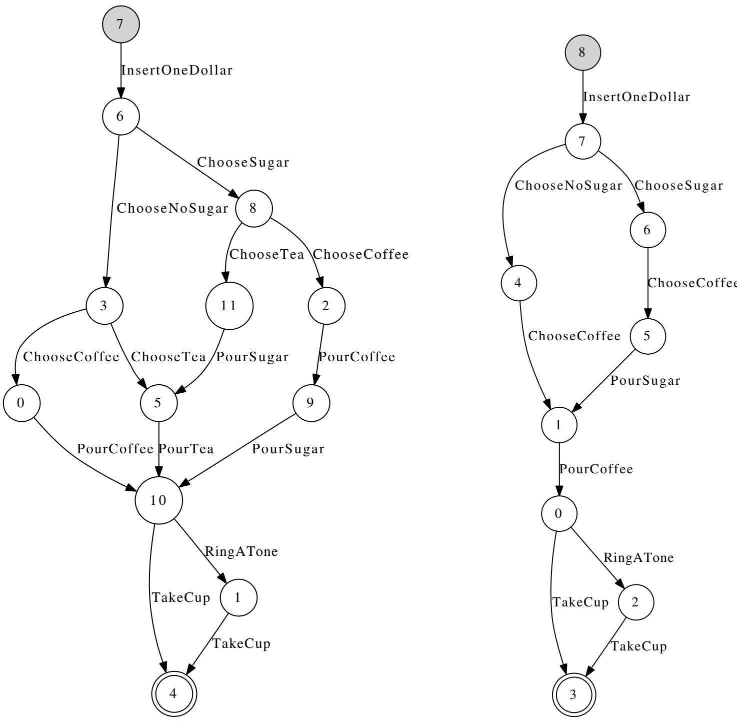


Fig. 5. Synthesized Canadian coffee machines (offering and not offering tea)

1. The framework of Classen et al. is based on a combination of Featured Transition Systems (FTS, [9]) and linear-time temporal logic (LTL, [50]). An FTS is a Labeled Transition System (LTS) that is associated with a feature diagram [51] and where individual transitions are "colored" according to the features to which they correspond. Products are obtained by pruning those transitions that are not involved according to the feature diagram. Their correctness is verified by applying model checking techniques to (sets of) products of the SPL. Therefore, the constraints to be checked have to be defined separately in the sense that they are not contained in the actual variability model.
2. The framework of Asirelli et al. combines Modal Transition Systems (MTS) [52] with a branching-time temporal logic (MHML [48]). An MTS is an LTS which distinguishes *may* and *must* transitions, which in the context of SPLs are used to model optional and mandatory software features. MHML is an extension of the classical Hennessy-Milner logic (HML, [53]) that achieves awareness of the different types of transitions in an MTS, and adds path quantifiers as known from other branching-time logics. As such, MHML complements the behavioral descriptions of MTS, as it allows to formulate static constraints that can not be expressed by an MTS.

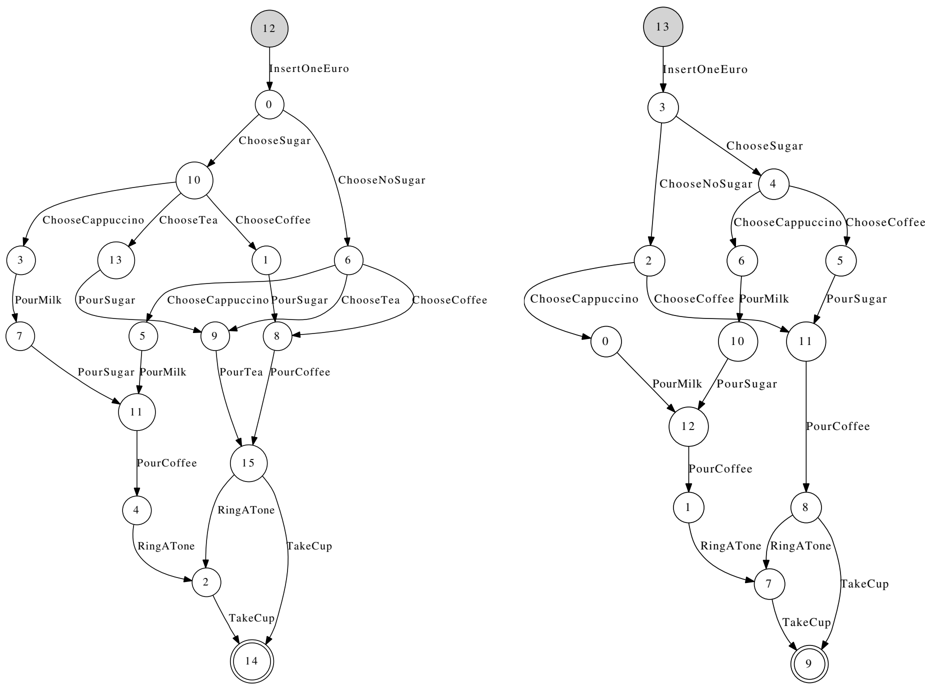


Fig. 6. Synthesized European coffee machines (offering and not offering tea)

To derive products, the framework basically follows a PoE (property-oriented expansion [54]) approach, pruning or enforcing may transitions in the MTS in a counter-example-guided way based on model checking of the MHML formulae of the variability model [48]. Correctness of this heuristics is enforced by continuous control of the refinement steps successively applied to the variability model via model checking. Completeness, on the other hand, is not guaranteed.

The whole MTS-based approach to derive products can be regarded as a special case of our synthesis-based approach to loose programming, as MTSs can directly be considered as specific loose programs. Another possible approach to achieve a complete and correct algorithm for the MTS setting is to apply synthesis algorithms for branching-time logics [10], as MTSs can be fully characterized along the lines of [55, 56] simply by omitting the exist clause for may transitions.

Figure 7 shows that these two approaches are based on the idea to flexibilize transition systems for the representation of variants. Still, these structures often require to overspecify the envisioned solution space, as, for example, even the flexibilized versions do not fully support disjunction, the perhaps most natural operator to express variability. Rather, they provide restricted means of choice, such as using the flexibility of may transitions, or explicit means of varying

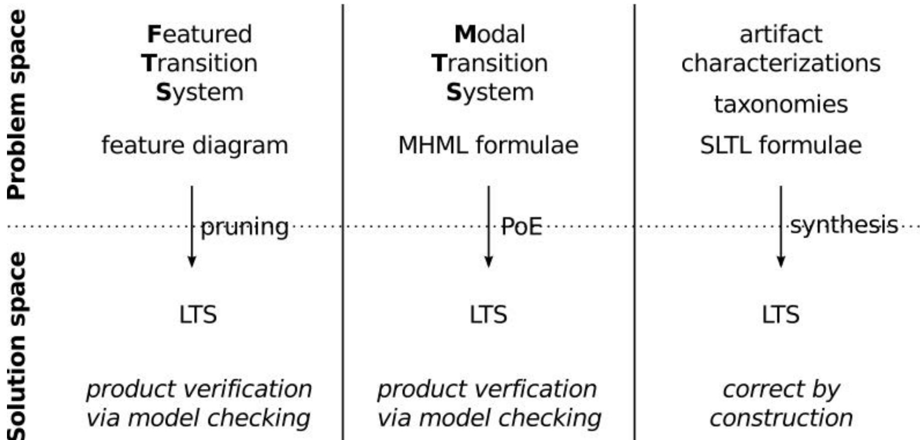


Fig. 7. Comparison of the three approaches

instantiations. This typically constrains the solution space more than necessary, while it, at the same time, does not allow to, for instance, model dependencies between instantiations. Thus, these variability models are often at the same time too restrictive and to lax. The latter problem is therefore addressed by subsequent model checking, validating the instantiation process, and the former problem by manual remodeling in response to the counterexamples provided by the model checker in case of failure.

Our constraint-based correctness-by-construction process avoids these problems, as its underlying constraint language allows one to fully describe the intended solution space without imposing any overspecification. The concretization to explicit products is then done via synthesis as has been extensively illustrated for the the coffee machine product family in the previous section. This change may in particular be regarded as a step from a setting with *closed-world assumption* to one with an *open-world assumption.*

5 Conclusion

With this paper, we demonstrated the power of combining modern process synthesis technology with a constraint-oriented approach to variability modeling. The point of this combination is that it guarantees the validity of all the required properties simply by construction: guaranteeing a new property simply requires adding a corresponding constraint. The synthesis procedure then automatically takes care that all generated variants are property-conform. This fully declarative approach leads to a very agile variability framework, where new product lines guaranteeing new properties can be defined ad hoc and are, due to our synthesis technology, immediately operational. The impact and ease of handling of our method has been illustrated along the coffee machine case study

of [48, 49], which has already been used by different groups to illustrate their approaches.

Some of the future work listed in [48] is readily covered by our approach:

– *"how to identify classes of properties that, once proved over family descriptions, are preserved by all products of the family"*: preservation of properties is automatically guaranteed in our framework,
– *"how to hide the complexity of the proposed modeling and verification framework from end users"*, along with future work listed in [49]: *"ideally, engineers should be able to use high-level languages hiding all semantic details"*: PROPHETS, our implementation of the synthesis approach, makes use of intuitive graphical formalisms and provides template formulae to ease the constraint-based specification. Based on this, it directly produces product variants as a means for further selection by the user.

The success of our approach depends on the quality of the by no means trivial domain modeling, which comprises the definition of the domain-specific vocabulary in terms of artifact and type taxonomies, artifact interface descriptions in terms of the vocabulary, and temporal-logic constraints for expressing both domain-wide rules for the adequate combination of artifacts, as well as intent, that is, the goal one wants to achieve with the product (lines). Whereas taxonomies and interface descriptions are commonly part of architectural descriptions in software engineering, temporal constraints are typical for approaches to behavior-oriented (variability) modeling. Our constraint-based variability modeling approach allows one to combine and exploit these architectural and behavioral descriptions without imposing any overspecification. This does not only concern the structure, but also the underlying artifact libraries, and therefore has the flavor of an *open-world assumption*.

The main bottleneck of our approach is scalability, both computationally, as the synthesis process is extremely costly, and concerning the solution space, as the number of proposed solutions may be gigantic. We are currently investigating numerous ways to overcome these problems. For instance, in [29] we illustrated how playing with constraints helps mastering the size of the solution space, and in [30] we investigated the impact of imposing structure in terms of loose programs in order to localize/decompose the synthesis process. The latter approach may be considered a hybrid, where ideas of [9] and [48] are integrated in a fashion already foreseen in [25].

References

1. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering - Foundations, Principles, and Techniques. Springer (2005)
2. Sinnema, M., Deelstra, S.: Classifying variability modeling techniques. Information and Software Technology 49(7), 717–739 (2006)
3. Czarnecki, K.: Variability Modeling: State of the Art and Future Directions. In: VaMoS, ICB-Research Report No. 37, University of Duisburg Essen, p. 11 (2010)

4. Chen, L., Babar, M.A.: A systematic review of evaluation of variability management approaches in software product lines. Information and Software Technology 53(4), 344–362 (2011)
5. Thüm, T., Schaefer, I., Kuhlemann, M., Apel, S.: Proof composition for deductive verification of software product lines. In: Proc. Int'l Workshop Variability-intensive Systems Testing, Validation and Verification (VAST), pp. 270–277. IEEE Computer Society (2011)
6. Lochau, M., Oster, S., Goltz, U., Schürr, A.: Model-based Pairwise Testing for Feature Interaction Coverage in Software Product Line Engineering. Software Quality Journal, 1–38 (2011)
7. Bruns, D., Klebanov, V., Schaefer, I.: Verification of Software Product Lines with Delta-Oriented Slicing. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 61–75. Springer, Heidelberg (2011)
8. Schaefer, I., Rabiser, R., Clarke, D., Bettini, L., Benavides, D., Botterweck, G., Pathak, A., Trujilol, S., Villela, K.: Software Diversity – State of the Art and Perspectives. STTT (2012)
9. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A., Raskin, J.F.: Model checking lots of systems: efficient verification of temporal properties in software product lines. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010), vol. 1, pp. 335–344. ACM, New York (2010)
10. Asirelli, P., ter Beek, M.H., Gnesi, S., Fantechi, A.: Deontic logics for modeling behavioural variability. In: VaMoS, Essen, Germany, pp. 71–76 (January 2009)
11. Apel, S., Kästner, C., Grösslinger, A., Lengauer, C.: Type safety for feature-oriented product lines. Automated Software Engineering 17(3), 251–300 (2010)
12. Kästner, C., Apel, S., Thüm, T., Saake, G.: Type checking annotation-based product lines. ACM Transactions on Software Engineering and Methodology (TOSEM) (to appear, 2012)
13. Cordy, M., Classen, A., Perrouin, G., Heymans, P., Schobbens, P.Y., Legay, A.: Simulation Relation for Software Product Lines: Foundations for Scalable Model Checking. In: Proceedings of 34th International Conference on Software Engineering (ICSE 2012). IEEE (to appear, 2012)
14. Hähnle, R., Schaefer, I.: A Liskov Principle for Delta-Oriented Programming. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012, Part I. LNCS, vol. 7609, pp. 32–46. Springer, Heidelberg (2012)
15. Liskov, B., Wing, J.M.: A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems (TOPLAS) 16(6), 1811–1841 (1994)
16. Delaware, B., Cook, W., Batory, D.: A Machine-Checked Model of Safe Composition. In: 8th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2009), pp. 31–35. ACM (2009)
17. Schaefer, I., Bettini, L., Damiani, F.: Compositional type-checking for delta-oriented programming. In: 10th International Conference on Aspect-Oriented Software Development (AOSD 2011), pp. 43–56. ACM (2011)
18. Schaefer, I., Lamprecht, A.L., Margaria, T.: Constraint-oriented Variability Modeling. In: Rash, J., Rouff, C. (eds.) 34th Annual IEEE Software Engineering Workshop (SEW-34), pp. 77–83. IEEE CS Press (June 2011)
19. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A.: Symbolic model checking of software product lines. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011), pp. 321–330. ACM, New York (2011)
20. Steffen, B., Margaria, T., Freitag, B.: Module Configuration by Minimal Model Construction. Technical report, Fakultät für Mathematik und Informatik, Universität Passau (1993)

21. Freitag, B., Steffen, B., Margaria, T., Zukowski, U.: An Approach to Intelligent Software Library Management. In: Proceedings of the 4th International Conference on Database Systems for Advanced Applications (DASFAA), pp. 71–78. World Scientific Press (1995)
22. Steffen, B., Margaria, T., Braun, V.: The Electronic Tool Integration platform: concepts and design. International Journal on Software Tools for Technology Transfer (STTT) 1(1-2), 9–30 (1997)
23. Steffen, B., Margaria, T., von der Beeck, M.: Automatic synthesis of linear process models from temporal constraints: An incremental approach. In: ACM/SIGPLAN Int. Workshop on Automated Analysis of Software (AAS 1997) (1997)
24. Steffen, B., Margaria, T., Claßen, A., Braun, V., Reitenspieß, M.: An Environment for the Creation of Intelligent Network Services. In: Intelligent Networks: IN/AIN Technologies, Operations, Services and Applications - A Comprehensive Report, IEC: International Engineering Consortium, pp. 287–300 (1996)
25. Steffen, B.: Method for incremental synthesis of a discrete technical system (1998)
26. Naujokat, S., Lamprecht, A.L., Steffen, B.: Tailoring Process Synthesis to Domain Characteristics. In: Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS) (2011)
27. Lamprecht, A.L., Margaria, T., Steffen, B.: Bio-jETI: a framework for semantics-based service composition. BMC Bioinformatics 10(suppl. 10), S8 (2009)
28. Lamprecht, A.L., Naujokat, S., Margaria, T., Steffen, B.: Semantics-based composition of EMBOSS services. Journal of Biomedical Semantics 2(suppl. 1), S5 (2011)
29. Lamprecht, A.L., Naujokat, S., Steffen, B., Margaria, T.: Constraint-Guided Workflow Composition Based on the EDAM Ontology. In: Burger, A., Marshall, M.S., Romano, P., Paschke, A., Splendiani, A. (eds.) Proceedings of the 3rd Workshop on Semantic Web Applications and Tools for Life Sciences (SWAT4LS 2010). CEUR Workshop Proceedings, vol. 698 (December 2010)
30. Lamprecht, A.L., Naujokat, S., Margaria, T., Steffen, B.: Synthesis-Based Loose Programming. In: Proceedings of the 7th International Conference on the Quality of Information and Communications Technology (QUATIC) (September 2010)
31. Naujokat, S., Lamprecht, A.-L., Steffen, B.: Loose Programming with PROPHETS. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 94–98. Springer, Heidelberg (2012)
32. Lamprecht, A.L., Margaria, T., Steffen, B., Sczyrba, A., Hartmeier, S., Giegerich, R.: GeneFisher-P: variations of GeneFisher as processes in Bio-jETI. BMC Bioinformatics 9(suppl. 4), S13 (2008)
33. Ebert, B.E.: A systems approach to understand and engineer whole-cell redox biocatalysts. Dissertation, Fakultät Bio- und Chemieingenieurwesen, Technische Universität Dortmund (2011)
34. Steffen, B., Margaria, T., Claßen, A., Braun, V., Nisius, R., Reitenspieß, M.: A Constraint-Oriented Service Creation Environment. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 418–421. Springer, Heidelberg (1996)
35. Steffen, B., Margaria, T.: MetaFrame in Practice: Design of Intelligent Network Services. In: Olderog, E.-R., Steffen, B. (eds.) Correct System Design. LNCS, vol. 1710, pp. 390–415. Springer, Heidelberg (1999)
36. Margaria, T., Steffen, B.: Aggressive Model-Driven Development: Synthesizing Systems from Models viewed as Constraints. In: MBEES, pp. 51–62 (2005)
37. Margaria, T., Steffen, B.: Agile IT: Thinking in User-Centric Models. In: Margaria, T., Steffen, B. (eds.) ISoLA 2008. CCIS, vol. 17, pp. 490–502. Springer, Heidelberg (2009)

38. Jörges, S., Lamprecht, A.L., Margaria, T., Schaefer, I., Steffen, B.: A Constraint-based Variability Modeling Framework. International Journal on Software Tools for Technology Transfer (STTT) (to appear, 2012)
39. Margaria, T., Steffen, B.: Business Process Modelling in the jABC: The One-Thing-Approach. In: Cardoso, J., van der Aalst, W. (eds.) Handbook of Research on Business Process Modeling. IGI Global (2009)
40. Steffen, B.: Unifying Models. In: Reischuk, R., Morvan, M. (eds.) STACS 1997. LNCS, vol. 1200, pp. 1–20. Springer, Heidelberg (1997)
41. Steffen, B., Rüthing, O.: Quality Engineering: Leveraging Heterogeneous Information (Invited Talk). In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 23–37. Springer, Heidelberg (2011)
42. Margaria, T., Steffen, B.: Service-Orientation: Conquering Complexity with XMDD. In: Hinchey, M., Coyle, L. (eds.) Conquering Complexity, pp. 217–236. Springer, London (2012)
43. Steffen, B., Margaria, T., Claßen, A., Braun, V.: Incremental Formalization: A Key to Industrial Success. Software - Concepts and Tools 17(2), 78–95 (1996)
44. Steffen, B., Margaria, T., Nagel, R., Jörges, S., Kubczak, C.: Model-Driven Development with the jABC. In: Bin, E., Ziv, A., Ur, S. (eds.) HVC 2006. LNCS, vol. 4383, pp. 92–108. Springer, Heidelberg (2007)
45. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The description logic handbook: theory, implementation, and applications. Cambridge University Press, New York (2003)
46. Schreiber, G., Dean, M.: OWL Web Ontology Language Reference. W3C Recommendation (2004), `http://www.w3.org/TR/owl-ref/` (last accessed June 25, 2012)
47. May, C.: Entwicklung einer Bibliothek zur service-orientierten Modellierung von Ontologien. Diploma thesis, TU Dortmund (2009)
48. Asirelli, P., ter Beek, M.H., Fantechi, A., Gnesi, S.: A Logical Framework to Deal with Variability. In: Méry, D., Merz, S. (eds.) IFM 2010. LNCS, vol. 6396, pp. 43–58. Springer, Heidelberg (2010)
49. Asirelli, P., ter Beek, M.H., Gnesi, S., Fantechi, A.: Formal Description of Variability in Product Families. In: 15th International Software Product Line Conference, SPLC 2011, pp. 130–139 (2011)
50. Clarke, E.M., Grumberg, O., Peled, D.A.: Temporal Logics. In: Model Checking, pp. 27–32. The MIT Press (1999)
51. Apel, S., Kästner, C.: An Overview of Feature-Oriented Software Development. Journal of Object Technology 8(5), 49–84 (2009)
52. Larsen, K., Thomsen, B.: A modal process logic. In: Proceedings of the Third Annual Symposium on Logic in Computer Science, LICS 1988, pp. 203–210 (July 1988)
53. Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. J. ACM 32, 137–161 (1985)
54. Steffen, B.: Property-Oriented Expansion. In: Cousot, R., Schmidt, D.A. (eds.) SAS 1996. LNCS, vol. 1145, pp. 22–41. Springer, Heidelberg (1996)
55. Steffen, B.: Characteristic Formulae. In: Ronchi Della Rocca, S., Ausiello, G., Dezani-Ciancaglini, M. (eds.) ICALP 1989. LNCS, vol. 372, pp. 723–732. Springer, Heidelberg (1989)
56. Steffen, B., Ingólfsdóttir, A.: Characteristic Formulae for Processes with Divergence. Information and Computation 110(1), 149–163 (1994)

Modeling Application-Level Management of Virtualized Resources in ABS*

Einar Broch Johnsen, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa

Department of Informatics, University of Oslo, Norway
{einarj,rudi,sltarifa}@ifi.uio.no

Abstract. Virtualization motivates lifting aspects of low-level resource management to the abstraction level of modeling languages, in order to model and analyze virtualized resource usage for application-level services and its relationship to service-level QoS. In this paper we illustrate how the modeling language ABS may be used for this purpose by modeling a service deployed on the cloud. Virtual machines are provided on demand to the service, which distributes service requests between its available machines depending on its application-level load balancing scheme. The resulting ABS models are used to relate the accumulated usage cost for the virtual machines to the obtained QoS for the service.

ABS is an abstract behavioral specification language for designing executable models of distributed object-oriented systems. The language combines advanced concurrency and synchronization mechanisms based on concurrent object groups with a functional language for modeling data. ABS supports deployment variability by dynamically created deployment components which act as resource-restricted execution contexts for ABS objects, for example with respect to CPU resources. The use of these artefacts is demonstrated in this paper through an example of service-level management of virtualized resources on the cloud.

1 Introduction

The abstract behavioral specification language ABS is a formal modeling language which aims at describing systems on a level that abstracts from implementation details but captures essential behavioral aspects of the targeted systems [21]. ABS targets the engineering of concurrent, component-based systems by means of executable object-oriented models which are easy to undertand for the software developer and allow rapid prototyping and analysis. The functional correctness of a targeted system largely depends on its high-level behavioral specification, independent of the platform on which the resulting code will be deployed. However, different deployment architectures may be envisaged for such a system, and the choice of deployment architecture may hugely influence the system's quality of service (QoS). For example, limitations in the processing capacity of the CPU of a cell phone may restrict the applications that can

* Partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu)

B. Beckert et al. (Eds.): FMCO 2011, LNCS 7542, pp. 89–108, 2013.
© Springer-Verlag Berlin Heidelberg 2013

be supported on the cell phone, and the capacity of a server may influence the response time for a service for peaks in the user traffic.

Recently, ABS has been extended with the concept of *deployment components* to capture the deployment architecture of target systems [24]. Whereas software components reflect the logical architecture of systems, deployment components reflect their deployment architecture. A deployment component is a resource-restricted execution context for a set of concurrent object groups, which controls how much computation can occur in this set between observable points in time. Deployment components may be dynamically created and they are parametric in the amount of resources they provide to their set of objects. This explicit representation of deployment architecture allows application-level response time and load balancing to be expressed in the system models in a very natural and flexible way, relative to the resources allocated to an application. This basic model of deployment architecture may be further extended by adding support for resource reallocation [23] or object mobility [25], allowing resources or objects to be moved from one deployment component to another.

The objective of this paper is to introduce and motivate the basic concepts developed to model deployment architecture in ABS in an informal way, rather than as a formal system. The formal syntax and semantics of ABS is discussed in [21] and the formalization of the integration of deployment components with ABS in [23–25]. In this paper we show how deployment components in ABS may be used to model virtualized systems, by developing an example inspired by cloud computing [8]. In this example, an abstract cloud provider offers virtual machines with given CPU capacities to client services, and bills the services based on an accounting scheme for their resource usage. The purpose of the model is not to make optimal use of hardware resources to provide these virtual machines (which would be the interest of the cloud provider) but rather to show how the developer may, at an early stage in the design of a service, gain insights into the resource needs of the service, and the trade-off between the cost of virtualized resources and the provided QoS for given customer scenarios.

The paper is structured as follows. Section 2 introduces timed ABS and shows how to model, e.g., response time in ABS models. Section 3 introduces the modeling concepts needed to capture deployment architecture and QoS in ABS. Section 4 introduces the case study developed in the paper and shows how the simulation tool for ABS can be used for rapid prototyping and analysis. Section 5 discusses related work and Section 6 concludes the paper.

2 Modeling Timed Behavior in ABS

ABS [21] is an executable object-oriented modeling language which targets distributed systems. The language is based on concurrent object groups, akin to concurrent objects (e.g., [9,12,22]), Actors (e.g., [1,19]), and Erlang processes [5]. A characteristic feature of concurrent object groups in ABS is that they internally support interleaved concurrency based on guarded commands. This makes it very easy to combine active and reactive behavior in the concurrent object

groups, based on cooperative scheduling of processes. A concurrent object group has at most one active process at any time and a queue of suspended processes waiting to execute on an object in the group. The processes stem from method activations. Objects in ABS are dynamically created from classes but typed by interface; i.e., there is no explicit notion of hiding as the object state is always encapsulated behind interfaces which offer methods to the environment. For simplicity in this paper, we do not use other code structuring mechanisms. This section informally reviews the core ABS language and its timed extension (for further details, see [7, 21]).

2.1 Core ABS

ABS is a modeling language which combines functional and imperative programming styles to develop high-level executable models. Concurrent object groups execute in parallel and communicate through asynchronous method calls. To intuitively capture internal computation inside a method, we use a simple functional language based on user-defined algebraic data types and functions. Thus, the modeler may abstract from the details of low-level imperative implementations of data structures, and still maintain an overall object-oriented design which is close to the target system. At a high level of abstraction, concurrent object groups typically consist of a single concurrent object; other objects may be introduced into a group as required to give some of the algebraic data structures an explicit imperative representation in the model. In this paper, we aim at high-level models and the groups will consist of single concurrent objects.

The *functional sublanguage* of ABS consists of a library of algebraic data types such as the empty type `Unit`, booleans `Bool`, integers `Int`, parametric data types such as sets `Set<A>` and maps `Map<A>` (given a value for the type variable `A`), and (parametric) functions over values of these data types. For example, we can define polymorphic sets using a type variable `A` and two constructors `EmptySet` and `Insert`, and a function `contains` which checks whether an element `el` is in a set `ss` recursively by pattern matching over `ss`:

```
data Set<A> = EmptySet | Insert(A, Set<A>);

def Bool contains<A>(Set<A> ss, A el) =
  case ss {
    EmptySet       => False ;
    Insert(el, _) => True;
    Insert(_, xs) => contains(xs, el);
  };
```

Here, the cases `p => exp` are evaluated in the listed order, underscore works as a wild card in the pattern `p`, and variables in `p` are bound in the expression `exp`.

The *imperative sublanguage* of ABS addresses concurrency, communication, and synchronization at the concurrent object level in the system design, and defines interfaces and methods with a Java-like syntax. ABS objects are *active*; i.e., their `run` method, if defined, gets called upon creation. *Statements* are standard for sequential composition $s_1; s_2$, assignments `x = rhs`, and for the **`skip`**, **`if`**, **`while`**, and **`return`** constructs. The statement **`suspend`** unconditionally

suspends the execution of the active process of an object by adding this process to the queue, from which an enabled process is then selected for execution. In **await** g, the guard g controls the suspension of the active process and consists of Boolean conditions b and return tests $x?$ (see below). Just like functional expressions e, guards g are side-effect free. If g evaluates to false, the active process is suspended, i.e., added to the queue, and some other process from the queue may execute. *Expressions* `rhs` include the creation of an object group **new cog** `C(e)`, object creation in the group of the creator **new** `C(e)`, method calls `o!m(e)` and `o.m(e)`, future dereferencing `x.`**get**, and pure expressions `e` apply functions from the functional sublanguage to state variables.

Communication and *synchronization* are decoupled in ABS. Communication is based on asynchronous method calls, denoted by assignments `f=o!m(e)` to future variables `f`. Here, `o` is an object expression and `e` are (data value or object) expressions providing actual parameter values for the method invocation. (Local calls are written **this**`!m(e)`.) After calling `f=o!m(e)`, the future variable `f` refers to the return value of the call and the caller may proceed with its execution *without blocking* on the method reply. There are two operations on future variables, which control synchronization in ABS. First, the guard **await** `f?` *suspends the active process* unless a return to the call associated with `f` has arrived, allowing other processes in the object group to execute. Second, the return value is retrieved by the expression `f.`**get**, which *blocks all execution in the object* until the return value is available. The statement sequence `x=o!m(e);v=x.`**get** encodes commonly used *blocking calls*, abbreviated `v=o.m(e)` (often referred to as synchronous calls). If the return value of a call is without interest, the call may occur directly as a statement `o!m(e)` with no associated future variable. This corresponds to message passing in the sense that there is no synchronization associated with the call.

2.2 Real-Time ABS

Real-Time ABS [7] is an extension of ABS which captures the timed behavior of ABS models. An ABS model is a model in Real-Time ABS in which execution takes zero time; thus, standard statements in ABS are assumed to execute in zero time. Timing aspects may be added incrementally to an untimed behavioral model. Our approach extends the distributed concurrent object groups in ABS with an integration of both *explicit* and *implicit* time.

Deadlines. The object-oriented perspective on timed behavior is captured by *deadlines* to method calls. Every method activation in Real-Time ABS has an associated deadline, which decrements with the passage of time. This deadline can be accessed inside the method body with the expression **deadline**`()`. Deadlines are *soft*; i.e., **deadline**`()` may become negative but this does not in itself block the execution of the method. By default the deadline associated with a method activation is infinite, so in an untimed model deadlines will never be missed. Other deadlines may be introduced by means of call-site *annotations*.

Real-Time ABS introduces two new data types into the functional sublanguage of ABS: `Time`, which has the constructor `Time(r)`, and `Duration`,

which has the constructors `InfDuration` and `Duration(r)`, where `r` is a value of the type `Rat` of rational numbers. The accessor functions `timeVal` and `durationValue` returns `r` for time and duration values `Time(r)` and `Duration(r)`, respectively. Let `o` be an object which implements a method `m`. Below, we define a method `n` which calls `m` on `o` and specifies a deadline for this call, given as an annotation and expressed in terms of its own deadline. Remark that if its own deadline is `InfDuration`, then the deadline to `m` will also be unlimited. The function `scale(d,r)` multiplies a duration `d` by a rational number `r` (the definition of `scale` is straightforward).

```
Int n (T x){ [Deadline: scale(deadline(),0.9)] return o.m(x); }
```

Explicit Time. In the explicit time model of Real-Time ABS [14], the execution time of computations is modeled using *duration statements* **`duration`**`(e1,e2)` with best- and worst-case execution times `e1` and `e2`. This is the standard approach to modeling timed behavior, known from, e.g., timed automata in UPPAAL [27]. These statements are inserted into the model, and capture execution time which does not depend on the system's deployment architecture. Let `f` be a function defined in the functional sublanguage of ABS, which recurses through some data structure `x` of type `T`, and let `size(x)` be a measure of the size of this data structure `x`. Consider a method `m` which takes as input such a value `x` and returns the result of applying `f` to `x`. Let us assume that the time needed for this computation depends on the size of `x`; e.g., the computation time is between a duration `0.5*size(x)` and a duration `4*size(x)`. An interface `I` which provides the method `m` and a class `C` which implements `I`, including the execution time for `m` using the explicit time model, are specified as follows:

```
interface I { Int m(T x) }
class C implements I {
  Int m (T x){ duration(0.5*size(x), 4*size(x)); return f(x);
  }
}
```

Implicit Time. In the implicit time model of Real-Time ABS, the execution time is not specified explicitly in terms of durations, but rather *observed* on the executing model. This is done by comparing clock values from a global clock, which can be read by an expression **`now`**`()` of type `Time`. We specify an interface `J` with a method `p` which, given a value of type `T`, returns a value of type `Duration`, and implement `p` in a class `D` such that `p` measures the time needed to call the method `m` above, as follows:

```
interface J { Duration p (T x) }
class D implements J (I o) {
   Duration p (T x){ Time start; Int y;
     start = now();  y=o.m(x); return timeDifference(now(),start);
   }
}
```

Observe that by using the implicit time model, no assumptions about execution times are specified in the model above. The execution time depends on how quickly the method call is effectuated by the called object. The execution time is simply measured during execution by comparing the time before and after

making the call. As a consequence, the time needed to execute a statement with the implicit time model depends on the *capacity* of the chosen deployment architecture and on *synchronization* with (slower) objects.

3 Modeling Deployment Architectures in ABS

3.1 Deployment Components

A *deployment component* in Real-Time ABS captures the execution capacity associated with a number of concurrent object groups. Deployment components are first-class citizens in Real-Time ABS, and provide a given amount of resources which are shared by their allocated objects. Deployment components may be dynamically created depending on the control flow of the ABS model or statically created in the main block of the model. We assume a deployment component `environment` with unlimited resources, to which the root object of a model is allocated. When objects are created, they are by default allocated to the same deployment component as their creator, but they may also be allocated to a different component. Thus, a model without explicit deployment components runs in `environment`, which does not impose any restrictions on the execution capacity of the model. A model may be extended with other deployment components with different processing capacities.

Given the interfaces `I` and `J` and classes `C` and `D` defined in Section 2.2, we can for example specify a deployment architecture in which two `C` objects are deployed on different deployment components `server1` and `server2`, and interact with the `D` objects deployed on a deployment component `clientServer`. Deployment components in Real-Time ABS have the type `DC` and are instances of the class `DeploymentComponent`. This class takes as parameters a name, given as a string, and a set of restrictions on resources. The name is mainly used for monitoring purposes. Here we focus on resources reflecting the components' processing capacity, which are specified by the constructor `CPUCapacity(r)`, where `r` represents the amount of abstract processing resources available between observable points in time. Below, we create three deployment components `Server1`, `Server2`, and `ClientServer`, with the processing capacities 6, 3, and unlimited (i.e., `ClientServer` has no resource restrictions). The local variables `server1`, `server2`, and `clientServer` refer to these three deployment components, respectively. Objects are explicitly allocated to the servers by annotations; below, `object1` is allocated to `Server1`, etc.

```
{ // This main block initializes a static deployment architecture:
  DC server1 = new DeploymentComponent("Server1",set[CPUCapacity(6)]);
  DC server2 = new DeploymentComponent("Server2",set[CPUCapacity(3)]);
  DC clientServer = new DeploymentComponent("ClientServer", EmptySet);
  [DC: server1] I object1 = new cog C;
  [DC: server2] I object2 = new cog C;
  [DC: clientServer] J client1monitor = new cog D(object1);
  [DC: clientServer] J client2monitor = new cog D(object2);
}
```

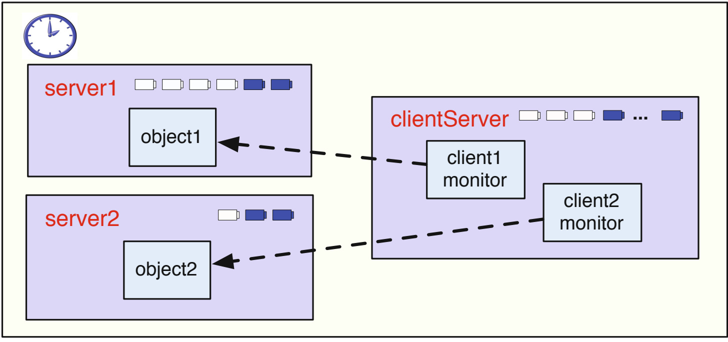


Fig. 1. A deployment architecture in Real-Time ABS, with three deployment components `server1`, `server2`, and `clientServer` (see Section 3.1). In each deployment component, we see its allocated objects and the "battery" of allocated and available processing resources (top right of each deployment component).

Figure 1 depicts this deployment architecture and the artefacts introduced into the modeling language. Since all objects are allocated to a deployment component (which is `environment` unless overridden by an annotation), we let the expression **`thisDC`**`()` evaluate to the deployment component of an object. For convenience, a call to the method `total("CPU")` of a deployment component returns its total amount of allocated CPU resources.

3.2 Resource Costs

The available resource capacity of a deployment component determines how much computation may occur in the objects allocated to that component. Objects allocated to the component compete for the shared resources in order to execute, and they may execute until the component runs out of resources or they are otherwise blocked. For the case of CPU resources, the resources of the component define its processing capacity between observable (discrete) points in time, after which the resources are renewed.

Cost Models. The cost of executing statements in the ABS model is determined by a default value which is set as a compiler option (e.g., `defaultcost=10`). However, the default cost does not discriminate between statements and we may want to introduce a more refined cost model. For example, if `e` is a complex expression, then the statement `x=e` should have a significantly higher cost than **`skip`** in a realistic model. For this reason, more fine-grained costs can be inserted into Real-Time ABS models by means of annotations. For example, let us assume that the cost of computing the function `f(x)` defined in Section 2.2 may be given as a function `g` which depends on the size of the input value `x`. In the context of deployment components, we may redefine the implementation of interface `I` above to be *resource-sensitive* instead of having a predefined duration as in the explicit time model. The resulting class `C2` can be defined as follows:

```
class C2 implements I {
  Int m (T x){ [Cost: g(size(x))] return f(x);
  }
}
```

It is the responsibility of the modeler to specify an appropriate cost model. A behavioral model with default costs may be gradually refined to provide more realistic resource-sensitive behavior. For the computation of the cost functions such as `g` in our example above, the modeler may be assisted by the COSTABS tool [2], which computes a worst-case approximation of the cost for `f` in terms of the input value `x` based on static analysis techniques, when given the ABS definition of the expression `f`. However, the modeler may also want to capture resource consumption at a more abstract level during the early stages of system design, for example to make resource limitations explicit before a further refinement of a behavioral model. Therefore, cost annotations may be used by the modeler to abstractly represent the cost of some computation which remains to be fully specified. For example, the class `C3` below represents a draft version of our method `m` in which the worst-case cost of the computation is specified although the function `f` has yet to be introduced:

```
class C3 implements I {
  Int m (T x){ [Cost: size(x)*size(x)] return 0;
  }
}
```

4 Case Study: Application-Level Management of Virtualized Resources

A common strategy for web applications these days, especially in early development and deployment, is to acquire the needed resources (server, storage, bandwidth) from a cloud infrastructure provider such as Amazon instead of purchasing server hardware and data center space. In that way, initial costs can be kept low while still keeping the flexibility to react quickly to demand growth [8]. In this case study, we develop a model of a web application which distributes user requests to a number of servers deployed on the cloud. To clarify terminology, we shall refer to the clients of the web service as *users*, and the clients of the cloud provider (such as the web service) as *clients*.

A cloud infrastructure provider leases *virtual servers* to its clients by the CPU hour. Typically, the client application can select different configurations with respect to virtualized resources such as processing capacity, memory size, etc. The cost of leasing a virtual server depends on the configuration of these virtual resources, and in particular on the processing capacity of the virtual server. To keep costs down, it is in the interest of the client application that virtual servers are kept running only when they are busy processing requests from users, and that they are stopped and returned to the cloud provider otherwise. Consequently, a cloud-enabled application will typically have a component which handles the management of virtualized resources at the application level. This component monitors the user demand, provisions servers as needed, and distributes user

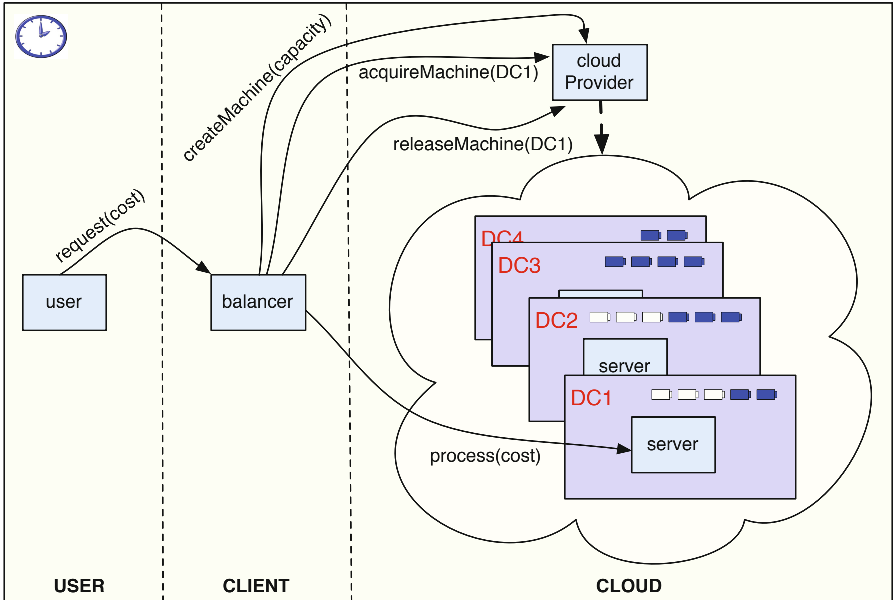


Fig. 2. An on-demand deployment architecture for the client application in Real-Time ABS. Neither user nor cloud provider contribute to the cost of running the system, and we assume the request processing costs dwarf the resources needed to run the balancer. Hence, only the servers are running in dedicated deployment components.

requests between the active servers in order to meet the deadlines of the user requests while keeping the costs of leasing virtual servers down.

In this section, we develop a Real-Time ABS model of a *client* application which interacts with a *cloud provider* and with a *user*. The model is depicted in Figure 2. This client application consists of a (dynamic) number of *servers* and one *balancer* which is the main focus of our case study. The balancer is in charge of the management of the virtualized resources acquired by the client application. The user sends processing requests to the balancer, which sends them to an active server. To keep the focus on the balancer, we do not model the details of these requests; instead, they carry a deadline and a processing cost that represent an abstraction of QoS and computing requirements. For the same reason, we do not provide the details of the cloud provider model in this paper.

It is the responsibility of the balancer to implement a resource management strategy which both minimizes the cost of running the client application on the cloud and maximizes the application's QoS (i.e., minimizes the number of deadline misses for user requests). Note that this model does not aim for precise measurements, but rather for a rough understanding of the system behavior. Hence, no precise costs of running the system are obtained via simulations (which would depend on the varying price of CPU hours). Rather, *different balancing strategies* can be compared by evaluation against different usage scenarios, for

```
interface CloudProvider {
  DC   createMachine(Int capacity);
  Unit acquireMachine(DC machine);
  Unit releaseMachine(DC machine);
  Int  getAccumulatedCost();
}

interface Balancer {
  Bool request (Int cost); // called from User
}

interface Server {
  Unit process(Int cost); // called from Balancer
  DC   getDC();
}
```

Fig. 3. Interfaces of the case study

example a user with a *steady request rate* or with an unexpected five-fold *load spike*.

Figure 3 shows the interfaces of the entities of the case study (the user needs no interface since it is not referenced by any object). Each `Server` has a method `process`, which incurs run-time costs on the server's deployment component, which can be found via the `getDC` method. The `Balancer`'s `request` method is called from the `User`. The balancer is responsible for creating `Server` objects on deployment components acquired from the `CloudProvider` via the method `createMachine`. Two methods `acquireMachine` and `releaseMachine` start and stop virtual machines (modeled by deployment components) so that the `Server` objects can process requests.

4.1 The Server and the Cloud Provider

The server and the cloud provider are implemented by two classes `Server` and `CloudProvider`, which do not change as we vary strategies and user behavior. The class `Server`, shown in Figure 4, implements the `Server` interface and is quite straightforward. The method `process` consumes resources according to its cost argument, and the method `getDC` simply returns the deployment component on which the server object is deployed.

The class `CloudProvider` implements the `CloudProvider` interface with methods for creating, acquiring and releasing virtual machines. This is done by creating deployment components on which the client application can deploy objects. In addition, the cloud provider keeps track of the accumulated costs incurred by the client application. The cost is calculated in terms of the sum of the processing capacities of the *active* virtual machines; i.e., a call to `acquireMachine(dc)` starts accounting for the virtual machine `dc` and a call to `releaseMachine(dc)` stops the accounting again for `dc`. The method `getAccumulatedCost` returns the accumulated cost of the client application. Inside the cloud provider, an active `run` method does the accounting for every time interval. Since our focus is the application-level management of virtualized

```
class Server implements Server {
    Unit process (Int cost) {
        while (cost > 0) { [Cost: 1] skip; cost = cost - 1; }
    }

    DC getDC() { return thisDC(); }
}
```

Fig. 4. Implementation of the `Server` class

resources, as implemented by the balancer, and not on specific strategies for cloud provisioning, we do not detail the cloud provider further in this paper.

4.2 The User Scenarios

We consider two user scenarios: *steady load* and *load spike*. The two scenarios are modeled by the corresponding classes `SteadyLoadUser` and `LoadSpikeUser`, given in Figure 5. The two classes have fields `numRequests` and `numFailures`, which are used for counting the number of sent requests and the number of missed deadlines for these requests, respectively. Both classes implement the method `sendRequest` which calls `request` with a given deadline on the balancer, suspends execution while waiting for the reply to the call, and does the bookkeeping after the reply has been received by incrementing the fields `numRequests` and `numFailures` as appropriate. The frequency of these requests is controlled by the active `run` method which differs between the two classes. In the `SteadyLoadUser` class, the `run` method asynchronously calls `sendRequest` and then suspends for a fixed duration. In contrast the `run` method of `LoadSpikeUser` has the same steady load behavior except for a window of time (between time 60 and 80 according to the clock), during which there is a load spike in which asynchronous calls to `sendRequest` are sent with much shorter intervals.

4.3 Balancing Strategies

In this case study, we model three different balancers for the application-level management of the virtualized resources. The balancers provide the front end to our web application, which receives user requests, and uses backend servers, deployed on the cloud, for processing these user requests. The different balancers reflect different strategies for interacting with the cloud provider to achieve the resource management, and may be described as follows:

- the **constant balancer** simply allocates one server sufficient for the expected load and keeps it running;
- the **as-needed balancer** calculates the server size needed to fulfill a specific request within the deadline, and allocates the needed resources disregarding the cost; and

```
class SteadyLoadUser(Balancer b) {
  Int numRequests = 0;
  Int numFailures = 0;
  Unit run() {
    while (True) {
      this!sendRequest();
      await duration(5, 5);
    }
  }
  Unit sendRequest() {
    [Deadline: Duration(2)] Fut<Bool> s = b!request(3);
    await s?; Bool success = s.get;
    numRequests = numRequests + 1;
    if (~success) numFailures = numFailures + 1;
  }
}
class LoadSpikeUser(Balancer b) {
  Int numRequests = 0;
  Int numFailures = 0;
  Unit run() {
      while (True) {
        if (timeVal(now()) > 60 && timeVal(now()) < 80) {
          this!sendRequest();
          await duration(1, 1);
        } else {
          this!sendRequest();
          await duration(5, 5);
        }
      }
    }
  Unit sendRequest() {
    [Deadline: Duration(2)] Fut<Bool> s = b!request(3);
    await s?; Bool success = s.get;
    numRequests = numRequests + 1;
    if (~success) numFailures = numFailures + 1;
  }
}
```

Fig. 5. Different user behavior modeled by the two classes `SteadyLoadUser` and `LoadSpikeUser`

– the **budget-aware balancer** operates with a given budget of CPU resources per time unit. Unused resources can be "saved for later" to cope with unexpected load spikes, but the cost of running the system is still bounded.

The Constant Balancer captures over-provisioning by processing all requests on a single server which should have sufficient capacity, and is modeled by the class `ConstantBalancer` in Figure 6. It initializes the web application by requesting a single machine from the cloud provider, on which it deploys a concurrent object group consisting of a `Server` object. After initialization, the constant balancer uses this server to process all user requests, and returns success to a user request if it was processed within the deadline.

```
class ConstantBalancer(CloudProvider provider, Int serverSize)
implements Balancer {
  Server server;
  DC dc;
  Bool initialized = False;
  Unit run() {
    Fut<DC> f = provider!createMachine(serverSize);
    await f?; dc = f.get;
    [DC: dc]server = new cog Server();
    initialized = True;
  }

  Bool request (Int cost) {
    await initialized;
    Fut<Unit> r = server!process(cost);
    await r?; return (durationValue(deadline()) > 0);
  }
}
```

Fig. 6. The Real-Time ABS model of the constant balancer

The As-Needed Balancer is modeled by the class `DynamicBalancer` in Figure 7. This class maintains a data structure `sleepingMachines` which sorts available machines (with deployed servers) by CPU processing capacity. We omit the (straightforward) definitions of the following auxiliary functions on this data structure: `hasMachine(s,i)` checks if a machine of capacity `i` is available in the structure `s`; `addMachine(s,i,m)` adds a machine `m` to the set associated with capacity `i` in `s`; and `removeMachine(s,i,m)` removes the machine `m` from the set associated with `i` in `s`.

When the `DynamicBalancer` receives a `request`, it calculates the machine capacity `resources` needed to fulfill the request, and requests a server deployed on a machine of appropriate size by calling **`this`**`.getMachine(resources)`. When it gets the server, it asynchronously calls `process` on this server and suspends. Once the reply is available, it calls **`this`**`.dropMachine(server)` and returns success to the user if the processing happened within the deadline.

The method `getMachine` first checks in `sleepingMachines` if there are available servers deployed on machines of appropriate size, in which case such a server is returned. (The auxiliary function `take(s)` selects an element of the set `s`.) Otherwise, the balancer requests a new machine from the cloud provider by calling `createMachine` and deploys a server on the new machine. The method `dropMachine` asks the cloud provider to stop running the machine on which the server is deployed and returns the server to the `sleepingMachines` set of appropriate capacity. The field `costPerTimeUnit` keeps track of the amount of resources *currently leased from the cloud provider*, and is updated by both methods `getMachine` and `releaseMachine`. This is the amount of resources for which the application is currently charged.

```
class DynamicBalancer(CloudProvider provider) implements Balancer {
  Map<Int, Set<Server>> sleepingMachines = EmptyMap;
  Int costPerTimeUnit = 0; Int machineStartTime = 0;

  Server getMachine(Int size) {
    Server server = null; Time t = now();
    costPerTimeUnit = costPerTimeUnit + size;
    if (hasMachine(sleepingMachines, size)) {
      server = take(lookup(sleepingMachines, size));
      sleepingMachines= removeMachine(sleepingMachines,size,server);
      Fut<DC> fdc = server!getDC();  await fdc?; DC dc = fdc.get;
      Fut<Unit> fa = provider!acquireMachine(dc); await fa?;
    } else {
      Fut<DC> fdc = provider!createMachine(size);
      await fdc?; DC dc = fdc.get;
      [DC: dc] server = new cog Server();
    }
    machineStartTime = timeDifference(t, now()); return server;
 }

 Unit dropMachine(Server server) {
   Fut<DC> fdc = server!getDC(); await fdc?; DC dc = fdc.get;
   Fut<Unit> fr = provider!releaseMachine(dc); await fr?;
   Fut<Int> fs = dc!total("CPU"); await fs?; Int size = fs.get;
   costPerTimeUnit = costPerTimeUnit - size;
   sleepingMachines = addMachine(sleepingMachines, size, server);
 }

 Bool request (Int cost) {
   Int resources = (cost / durationValue(deadline())) + 1
       + machineStartTime;
  Server server = this.getMachine(resources);
   Fut<Unit> r = server!process(cost);  await r?;
   this.dropMachine(server); return durationValue(deadline()) > 0;
 }
}
```

Fig. 7. The Real-Time ABS model of the as-needed balancer

The Budget-Aware Balancer is a resource management strategy in which the balancer has a certain *budget per time interval*, and may save resources for later. This balancer is modeled by the class `BudgetBalancer` in Figure 8, with a class parameter `budgetPerTimeUnit` which determines this budget, and a field `availableBudget` which keeps track of the accumulated (saved) resources. The fields `sleepingMachines`, `costPerTimeUnit`, and `machineStartTime` and the methods `getMachine` and `dropMachine` are as in the `DynamicBalancer` class. When the budget-aware balancer gets a request, it calculates the resources needed to fulfill the request in the variable `wantedResources` and the resources it has available on the budget in `maxResources`. If there are resources available on the budget, the budget-aware balancer calls `getMachine` to get the best server the request according to the budget. The budget-aware balancer has an active `run` method which monitors the resource usage and updates the available budget for every time interval. It also maintains a log `budgetHistory` of the available resources over time.

```
class BudgetBalancer(CloudProvider provider,Int budgetPerTimeUnit)
  implements Balancer {
  Map<Int, Set<Server>> sleepingMachines = EmptyMap;
  Int costPerTimeUnit = 0; Int machineStartTime = 0;

  Int availableBudget = 1;
  List<Int> budgetHistory = Nil;

  Unit run() {
   while (True) {
     availableBudget = availableBudget + budgetPerTimeUnit
         - costPerTimeUnit;
     budgetHistory = Cons(availableBudget, budgetHistory);
     await duration(1, 1);
     }
   }

  Bool request(Int cost) {
   Bool result = False;
   Int wantedResources = (cost / durationValue(deadline())) + 1
      + machineStartTime;
   Int maxResources = (budgetPerTimeUnit - costPerTimeUnit)
      + (max(availableBudget, 0) / durationValue(deadline()));
   if (maxResources > 0)  {
    Server server= this.getMachine(min(wantedResources,maxResources));
    Fut<Unit> r = server!process(cost);
    await r?;
    this.dropMachine(server);
    result = (durationValue(deadline()) > 0);
    }
   return result;
  }

  Server getMachine(Int size) { ... } // as in the DynamicBalancer
  Unit dropMachine(Server server) { ... }// as in the DynamicBalancer
}
```

Fig. 8. The Real-Time ABS model of the budget-aware balancer

4.4 Comparing Balancing Strategies

Real-Time Maude has a formally defined semantics [7] which is used to implement a model simulator in the Maude system [11]. In order to compare the three balancing strategies of our case study, we simulate their behavior for the two user scenarios described in Section 4.2, in each case with a single "user" object generating requests. For simplicity, we here set the budget of the budget-aware balancer to 1. All simulations were run for 100 units of simulated time. The following measurements were extracted from the simulation traces:

- *quality of service* measured as the number of successful requests (i.e., requests completed within the deadline) divided by the total number of requests; and
- *accumulated cost* of running the machines, measured as the total sum of CPU resources made available by the cloud provider.

Table 1 summarizes the results. Not surprisingly, the as-needed balancer exhibits the best QoS numbers, but at potentially unbounded runtime cost. The constant

Table 1. Simulation results

	User scenario			
	Steady load		*Load spike*	
Strategy	QoS	Cost	QoS	Cost
Constant balancer	100%	200	53%	200
As-needed balancer	100%	80	100%	128
Budget-aware balancer	100%	80	68%	97

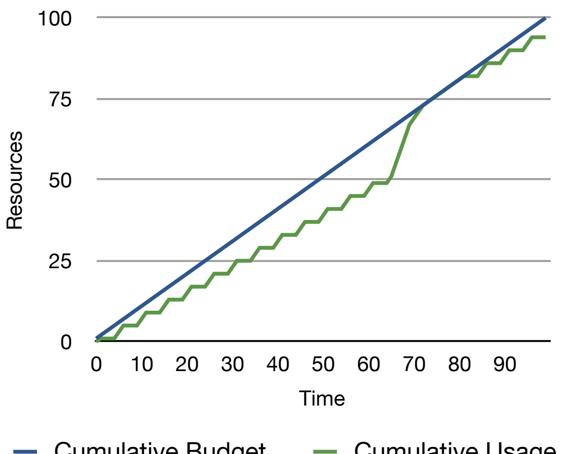


Fig. 9. Budget use over time for the budget-aware balancer. The load spike between time 60 and 80 quickly consumes the saved-up funds.

balancer with a single running server exhibited both the highest runtime cost and the worst QoS under unexpected load with the chosen scenarios.

The budget-aware strategy exhibits only slightly better QoS characteristics under load than the constant balancer approach, which reflects how the budget was chosen. Figure 9 shows the available and used budget over time. It can be seen that the available budget is mostly used during normal load, so there are not many saved resources which can be used to deal with the load spike between time 60 and 80. A more realistic system would have a monitoring component to alert an operator, who would be able to manually add budget or switch to other balancing strategies, but this functionality was not considered in our case study.

5 Related Work

The concurrency model of ABS is akin to concurrent objects and Actor-based computation, in which software units with encapsulated processors communicate asynchronously [5, 19, 22, 30]. Their inherent compositionality allows concurrent objects to be naturally distributed on different locations, because only the local state of a concurrent object is needed to execute its methods. In previous

work [4,23,24], the authors have introduced *deployment components* as a modeling concept for deployment architectures, which captures restricted resources shared between a group of concurrent objects, and shown how components with parametric resources may be used to capture a model's behavior for different assumptions about the available resources. The formal details of this approach are given in [24]. In previous work, the cost of execution was fixed in the language semantics. In this paper, we generalize that approach by proposing the specification of resource costs as part of the software development process. This is supported by letting default costs be overridden by annotations with user-defined cost expressed in terms of the local state and the input parameters to methods. This way, the cost of execution in the model may be adapted by the modeler to a specific cost scenario. This allows us to abstractly model the effect of deploying concurrent objects on deployment components with different amounts of allocated resources at an early stage in the software development process, before modeling the detailed control flow of the targeted system. In two larger case studies addressing resource management in the cloud [13,26], the presented approach is compared to specialized simulation tools and to measurements on deployed code.

Complementing the balancing strategies considered in this paper, the authors have studied extensions to the deployment component framework which support more advanced (or fine-grained) load-balancing. We have considered two such extensions, based on adding an expression `load(n)` which returns the average load of the current deployment component over the last `n` time intervals. First, by including resources as first-class citizens of ABS and allowing (virtual) resources to be reallocated between deployment components [23]. Second, by allowing objects to be marshaled and reallocated between deployment components [25].

Techniques for prediction or analysis of non-functional properties are based on either *measurement* or *modeling*. Measurement-based approaches apply to existing implementations, using dedicated profiling or tracing tools like JMeter or LoadRunner. Model-based approaches allow abstraction from specific system intricacies, but depend on parameters provided by domain experts [16]. A survey of model-based performance analysis techniques is given in [6]. Formal systems using process algebra, Petri Nets, game theory, and timed automata have been used in the embedded software domain (e.g., [10,17]), but also to the schedulability of processes in concurrent objects [20]. The latter work complements ours as it does not consider restrictions on shared deployment resources, but associates deadlines with method calls with abstract duration statements.

Work on modeling object-oriented systems with resource constraints is more scarce. Eckhardt et al. [15] use statistical modeling of meta-objects and virtual server replication to maintain service availability under denial of service attacks. Using the UML SPT profile for schedulability, performance, and time, Petriu and Woodside [28] informally define the Core Scenario Model (CSM) to solve questions that arise in performance model building. CSM has a notion of resource context, which reflects an operation's set of resources. CSM aims to bridge the gap between UML and techniques to generate performance models [6]. Closer to our

work is M. Verhoef's extension of VDM++ for embedded real-time systems [29], in which static architectures are explicitly modeled using CPUs and buses. The approach uses fixed resources targeting the embedded domain, namely processor cycles bound to the CPUs, while we consider more general resources for arbitrary software. Verhoef's approach is also based on abstract executable modeling, but the underlying object models and operational semantics differ. VDM++ has multi-thread concurrency, preemptive scheduling, and a strict separation of synchronous method calls and asynchronous signals, in contrast to our work with concurrent objects, cooperative scheduling, and caller-decided synchronization.

Others interesting lines of research are static cost analysis (e.g., [3, 18]) and symbolic execution for object-oriented programs. Most tools for cost analysis only consider sequential programs, and assume that the program is fully developed before cost analysis can be applied. COSTABS [2] is a cost analysis tool for ABS which supports concurrent object-oriented programs. Our approach, in which the modeler specifies cost in annotations, could be supported by COSTABS to automatically derive cost annotations for the parts of a model that are fully implemented. In collaboration with Albert *et al.*, we have applied this approach to memory analysis for ABS models [4]. However, the generalization of that work for general, user-defined cost models and its integration into the software development process remains future work. A future extension of our approach with symbolic execution would allow us to calculate best- and worst-case response time for the different balancing strategies depending on the available resources and the user load.

6 Conclusion

This paper gives an overview of how deployment architectures can be modeled by means of deployment components in Real-Time ABS. We show how this approach may be used to model virtualized systems by developing a case study of application-level management of virtualized resources in a cloud computing context. In our case study, an abstract cloud provider leases virtual machines with given amounts of CPU processing capacities to client services. The case study takes the *client perspective* on virtualized resource management, and models a client application for which three different proposals for a balancer class are compared in order to gain insights into their resource needs. The resulting models in Real-Time ABS are simulated for different user scenarios. In these scenarios, the cost of leasing resources from the cloud provider with the different resource management strategies are compared with respect to the QoS of the service for user requests. We are not aware of similar work addressing the formal modeling of application-level management of virtualized resources for cloud computing. However, we believe this problem is of increasing importance in a world of cloud-enabled applications.

References

1. Agha, G.A.: ACTORS: A Model of Concurrent Computations in Distributed Systems. The MIT Press (1986)
2. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: COSTABS: a cost and termination analyzer for ABS. In: Proc. Workshop on Partial Evaluation and Program Manipulation (PEPM 2012), pp. 151–154. ACM (2012)
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Java Bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007)
4. Albert, E., Genaim, S., Gómez-Zamalloa, M., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Simulating Concurrent Behaviors with Worst-Case Cost Bounds. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 353–368. Springer, Heidelberg (2011)
5. Armstrong, J.: Programming Erlang. Pragmatic Bookshelf (2007)
6. Balsamo, S., Marco, A.D., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: A survey. IEEE Transactions on Software Engineering 30(5), 295–310 (2004)
7. Bjørk, J., de Boer, F.S., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: User-defined schedulers for real-time concurrent objects. To Appear in Innovations in Systems and Software Engineering (2012)
8. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Computer Systems 25(6), 599–616 (2009)
9. Caromel, D., Henrio, L.: A Theory of Distributed Object. Springer (2005)
10. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Resource Interfaces. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 117–133. Springer, Heidelberg (2003)
11. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude. LNCS, vol. 4350. Springer, Heidelberg (2007)
12. de Boer, F.S., Clarke, D., Johnsen, E.B.: A Complete Guide to the Future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
13. de Boer, F.S., Hähnle, R., Johnsen, E.B., Schlatte, R., Wong, P.Y.H.: Formal Modeling of Resource Management for Cloud Architectures: An Industrial Case Study. In: De Paoli, F., Pimentel, E., Zavattaro, G. (eds.) ESOCC 2012. LNCS, vol. 7592, pp. 91–106. Springer, Heidelberg (2012)
14. de Boer, F.S., Jaghoori, M.M., Johnsen, E.B.: Dating Concurrent Objects: Real-Time Modeling and Schedulability Analysis. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 1–18. Springer, Heidelberg (2010)
15. Eckhardt, J., Mühlbauer, T., AlTurki, M., Meseguer, J., Wirsing, M.: Stable Availability under Denial of Service Attacks through Formal Patterns. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 78–93. Springer, Heidelberg (2012)
16. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by run-time parameter adaptation. In: Proc. ICSE, pp. 111–121. IEEE (2009)
17. Fersman, E., Krcál, P., Pettersson, P., Yi, W.: Task automata: Schedulability, decidability and undecidability. Information and Computation 205(8), 1149–1172 (2007)

18. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: Speed: Precise and Efficient Static Estimation of Program Computational Complexity. In: POPL, pp. 127–139. ACM (2009)
19. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theoretical Computer Science 410(2-3), 202–220 (2009)
20. Jaghoori, M.M., de Boer, F.S., Chothia, T., Sirjani, M.: Schedulability of asynchronous real-time concurrent objects. Journal of Logic and Algebraic Programming 78(5), 402–416 (2009)
21. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A Core Language for Abstract Behavioral Specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
22. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and Systems Modeling 6(1), 35–58 (2007)
23. Johnsen, E.B., Owe, O., Schlatte, R., Tapia Tarifa, S.L.: Dynamic Resource Reallocation between Deployment Components. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 646–661. Springer, Heidelberg (2010)
24. Johnsen, E.B., Owe, O., Schlatte, R., Tapia Tarifa, S.L.: Validating Timed Models of Deployment Components with Parametric Concurrency. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 46–60. Springer, Heidelberg (2011)
25. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: A Formal Model of Object Mobility in Resource-Restricted Deployment Scenarios. In: Arbab, F., Ölveczky, P. (eds.) FACS 2011. LNCS, vol. 7253, pp. 187–204. Springer, Heidelberg (2012)
26. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Modeling Resource-Aware Virtualized Applications for the Cloud in Real-Time ABS. In: Aoki, T., Tagushi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 71–86. Springer, Heidelberg (2012)
27. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer 1(1-2), 134–152 (1997)
28. Petriu, D.B., Woodside, C.M.: An intermediate metamodel with scenarios and resources for generating performance models from UML designs. Software and System Modeling 6(2), 163–184 (2007)
29. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 147–162. Springer, Heidelberg (2006)
30. Welc, A., Jagannathan, S., Hosking, A.: Safe futures for Java. In: Proc. OOPSLA, pp. 439–453. ACM Press (2005)

HATS Abstract Behavioral Specification: The Architectural View*

Reiner Hähnle[1], Michiel Helvensteijn[2], Einar Broch Johnsen[3], Michael Lienhardt[4], Davide Sangiorgi[4], Ina Schaefer[5], and Peter Y.H. Wong[6]

[1] Dept. of Computer Science, TU Darmstadt
haehnle@cs.tu-darmstadt.de
[2] CWI Amsterdam
Michiel.Helvensteijn@cwi.nl
[3] Dept. of Informatics, Univ. of Oslo
einarj@ifi.uio.no
[4] Dept. of Computer Science, Univ. of Bologna
{Davide.Sangiorgi,lienhard}@cs.unibo.it
[5] Dept. of Computer Science, TU Braunschweig
i.schaefer@tu-braunschweig.de
[6] Fredhopper B.V, Amsterdam
peter.wong@fredhopper.com

Abstract. The Abstract Behavioral Specification (ABS) language is a formal, executable, object-oriented, concurrent modeling language intended for behavioral modeling of complex software systems that exhibit a high degree of variation, such as software product lines. We give an overview of the architectural aspects of ABS: a feature-driven development workflow, a formal notion of deployment components for specifying environmental constraints, and a dynamic component model that is integrated into the language. We employ an industrial case study to demonstrate how the various aspects work together in practice.

1 Introduction

This is the third in a series of reports which together give a comprehensive overview of the possibilities and use cases of the Abstract Behavioral Specification (ABS) language developed within the FP7 EU project HATS (for "Highly Adaptable and Trustworthy Software using Formal Models"). Paper [18] describes the core part of ABS and its formal semantics while the tutorial [7] is about the modeling of variability in ABS using features and deltas. Delta-oriented programming [32] is a feature-oriented code reuse concept that is employed in HATS ABS as an alternative to traditional inheritance-based reuse.

The current paper is focussed on *architectural* aspects of modeling with ABS. A very brief summary of the main ideas of ABS is contained in Sect. 2 below.

* Partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu)

B. Beckert et al. (Eds.): FMCO 2011, LNCS 7542, pp. 109–132, 2013.
© Springer-Verlag Berlin Heidelberg 2013

In Sect. 3 we give a step-by-step guide on how to create a software product line in ABS from scratch using delta modeling. We make use of *abstract delta modeling* [8], dealing with conflicts explicitly. In Sect. 4 we turn to the question of how the deployment architecture of a system can be represented in a suitably platform-independent manner at the level of abstract models [19]. Sect. 5 reports on the component model used in HATS which makes it possible to align ABS models with architectural languages and opens the possibility of dynamic reconfiguration [28]. Finally, in Sect. 6 we tell—in the context of an industrial case study [35]—how the concepts discussed in this paper were shaped by application concerns and what the industrial prospects of HATS ABS are. This includes an account on how to unit test ABS models using the ABSUnit framework [14].

2 Abstract Behavioral Specification

The ABS language is designed for formal modeling and specification of concurrent, component-based systems at a level that abstracts away from implementation details, but retains essential behavioral and even deployment aspects. ABS follows a layered approach (see Fig. 1): at the base are functional abstractions around a standard notion of parametric algebraic data types. Next we have a OO-imperative layer similar to (but much simpler than) Java. The concurrency model of ABS is two-tiered: at the lower level, so called COGs (Concurrent Object Groups) encapsulate synchronous, multi-threaded, shared state computation on a single processor with cooperative scheduling. On top of this is an actor based model with asynchronous calls, message passing, active waiting, and future types. A syntactic module concept and assertions (including pluggable type systems) completes what we call *core ABS*. This language is described in detail in [18].

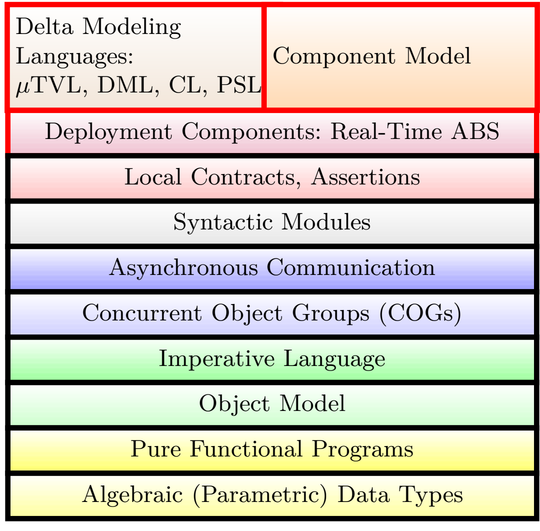


Fig. 1. Layered Architecture of ABS

ABS classes do not support code inheritance and don't define types. Management of code reuse is, instead, realized by *code deltas*, described in [7]. These are named entities that describe the code changes associated to realization of new features. The result is a separation of concerns between architecural/design issues and algorithmic/data type aspects. It helps early prototyping and avoids a disconnect between a system's architecture and its implementation.

In Sect. 3 we present the *delta modeling workflow* and demonstrate how it is used to implement software with a high degree of variability such as product lines.

Model-based approaches such as HATS face the challenge that, to be realistic, software models must address deployment issues, such as real-time requirements, capacity restrictions, latency, etc. *Real-time ABS*, introduced in Sect. 4, uses an additional language layer called *deployment component* to achieve this.

To achieve flexible dynamic behavior, but also to structure and encapsulate the dependencies in a software system, a formal notion of logical *component* is essential. To this end, ABS features a *component model*, which is orthogonal to delta modeling and presented in Sect. 5. Whereas *deployment components* are used to identify the deployment structure of the modeled system in terms of locations, logical *components* are used to identify the logical structure of the system in terms of units of behavior. In particular, a logical component may be distributed over several deployment components and several logical components may be (partly) located in the same deployment component.

We stress that all ABS language constructs have a formal semantics, details of which can be found in the technical deliverables of the HATS project [11–13]. In addition, ABS has been designed with an eye on analysability. A wide variety of tools for simulation, testing, resource estimation, safety analysis, and verification of ABS models are available.

3 The Delta Modeling Workflow

Variability at the level of abstract behavioral specifications (or source code) is represented in the ABS using the concept of delta modeling. Delta modeling was introduced by Schaefer et al. [31,32] as a novel modeling and programming language approach for software-based product lines. It can be seen as an alternative to feature-oriented programming [3]. Both approaches aim at automatically generating software products for a given valid collection of features, providing flexible and modular techniques to build different products that share functionality or code. In this section, we describe briefly how delta modeling is instantiated in ABS to represent software product lines. A more detailed account is [7]. We also introduce the *delta modeling workflow* for ABS, a step-by-step guide to building a product line, which leverages the flexibility of delta modeling.

3.1 Delta Modeling in ABS

Variability of software product lines at the requirements level is predominantly represented in terms of product features, where a feature is a user-visible product characteristic or an increment to functionality. A feature model [22] provides the set of possible product variants by associating them with the set of realized features. Features at this modeling level are merely names.

Delta modeling is a flexible, yet modular approach to implement different product variants by reusable artifacts. In delta modeling, the realization of a

software product line is divided into a core module and a set of delta modules. The core module implements the functionality common to all products of the software product line. Delta modules encapsulate modifications to the core product in order to realize other products. The modifications can include additions and removals of product entities and modifications of entities that are hierarchically composed. A particular product variant can be automatically derived by applying the modifications of a selected subset of the given delta modules to the core product. Which delta modules have to be applied for a specific product variant is based on the selection of desired features for this product variant. In order to automate this selection, each delta is associated with an application condition that is a Boolean constraint over the features in the feature module. If the constraint is satisfied for a specific feature selection, the respective delta has to be applied to generate the associated product variant. To avoid or resolve conflicts when two delta modules modify the same product entity in an incompatible manner, delta module application can be partially ordered. This also ensures that for any feature selection a uniquely defined product is generated.

The ABS incarnation of delta modeling is based on four languages (μTVL, DML, CL, PSL) which are defined on top of core ABS (see Fig. 1). The feature description language μTVL is used to describe the variability of a product line in terms of features and their attributes. At this level of abstraction, a feature is a name representing user-visible system functionality. Attributes represent micro-variability within features. μTVL is a textual description language for feature models and intended as a basis for other, e.g., graphical modeling formalisms.

The delta modeling language DML is used to specify delta modules containing modifications of a core ABS model. A delta module in ABS can add and remove classes and add as well as remove interfaces that are implemented by a class. Additionally, a delta module can modify the internal structure of classes by adding and removing fields and methods. Methods can also be modified by overriding the method body or by wrapping the previous implementation of the method using the `original()` call. Deltas can also be parameterized by specific values. Parameters are instantiated with concrete values during product generation, e.g., with the value that is set for a feature attribute.

The configuration language CL links μTVL feature models with the DML delta modules that implement the corresponding behavioral modifications. A CL specification provides application conditions for delta modules. They determine to which feature configurations modules are applied to in a `when` clause and provide an order to resolve conflicts between deltas modifying the same model entities The application ordering is given in `after` clauses attached to deltas stating that the delta must be applied after the given deltas if these are also applied during product generation.

The product selection language PSL is used to define the actual product of an ABS product line. A PSL script corresponds to a particular product variant and consists of two parts, namely, a specification of the features and their attributes selected for a product and an initialization block, which is often merely a call to an appropriate *main* method, even though it may contain configuration code.

To generate the product specified by the PSL script, all deltas with a valid application condition for the given feature selection are applied to the core ABS model in some order compliant with the order specified in the CL script. Finally, the initialization block is added to the core program.

3.2 Delta Modeling Workflow in ABS

Abstract Delta Modeling [8, 9] lends itself particularly well to a systematic development workflow for software product lines. One such workflow, dubbed *delta modeling workflow* [16, 17], was adapted to ABS. In the following we describe the workflow specifically with ABS in mind, which has not been done before. It is also used in the case study in Sect. 6.

The workflow gives step-by-step instructions for development of a software product line from scratch, directing developers to satisfy local constraints (more formally given in [16]) to guarantee desirable properties for the whole product line. These properties are described in Sect. 3.3.

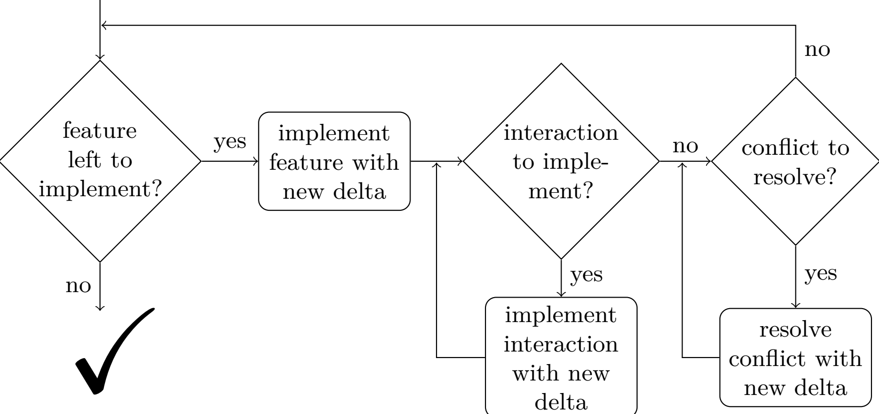


Fig. 2. Overview of the development workflow

A product line should be developed based on a product line specification, consisting of a feature model (which should include at least a subfeature hierarchy), and either formal or informal descriptions of each feature. When we start following the workflow, we assume that such a specification exists.

Briefly, features are implemented as a linear extension of the subfeature hierarchy. Base features are implemented first, subfeatures later, with one delta for each feature. Then, for every set of implemented features that should interact, but do not, we implement that interaction with a delta. Next, for every two deltas whose implementations are in conflict, a conflict resolving delta is written to resolve that issue. Fig. 2 shows a flow-chart that summarizes this process.

It is often suitable to put code common to all products into the core product. In the case of ABS, this means at least the following:

```
class Main { Unit run() {} }
{ new Main(); }
```

We start with a `Main` class with an empty `run` method. The second line creates a new `Main` instance, implicitly calling the `run` method, which will later be modified by deltas. It is possible to put mandatory features into the core product, but it is recommended that all features are implemented by deltas, as this makes the product line more robust to evolution [33], and promotes separation of concerns.

Also, we begin with a minimal ABS product line configuration: the list of features and the list of desired products, which can be empty.

```
productline name {
  features d_1, d_2, ..., d_n; }
```

In the following workflow description, we will use a subset of the Editor product line example, a complete version of which may be found in [9]. It describes a set of code editor widgets, as may be found in integrated development environments. The feature model of the Editor product line is shown in Fig. 3. Now we walk through the Delta Modeling Workflow depicted in Fig. 2.

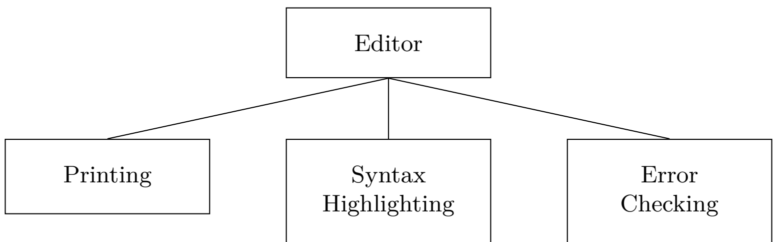


Fig. 3. Feature model of the Editor product line

Feature Left to Implement? In this stage of the workflow, we choose the next feature to implement. Essentially we walk through the subfeature hierarchy of the feature model in a topological order, i.e., base features first, subfeatures later. If all features have been implemented, we are finished.

For the example, we would have to start with the `Editor` feature. Any of the three features on the second level may be chosen next.

Implement Feature with New Delta. Having chosen a feature f, we now write a "feature delta" d_f to implement it:

```
delta d_f { ... }
```

The delta may add, remove or modify any classes and methods necessary to realize the functionality of f, while preserving the functionality of all superfeatures.

The developer only has to consider the feature-local code: the core product and the deltas implementing superfeatures of f. We now show the four feature deltas of the Editor product line (we leave out some details for the sake of brevity):

```
delta D_Editor {
  adds class Model { ... }
  adds class Font {
    Unit setColor(Color c) { ... }
    Color getColor() { ... }
    Unit setUnderlined(bool u) { ... }
    bool getUnderlined() { ... }
  }
  adds class Editor { Model model;
    { model = new Model(); ... }
    Model getChar(int c) {
      return model.getChar(c);
    }
    Font getFont(int c) {
      return new Font();
    } }
  modifies class Main {
    modifies Unit run() {
       new Editor();
    } } }
```

```
delta D_Printing {
  modifies class Editor {
    adds Unit print() {
      // print the plain text
    } } }
```

```
delta D_SyntaxHighlighting {
  adds class Highlighter(Model m) {
    Model model;
    { model = m; }
    Color getColor(int c) { ... }
  }
  modifies class Editor {
    modifies getFont(int c) {
      Font f = D_Editor.original(c);
      Highlighter h =
         new Highlighter(getModel());
      f.setColor(h.getColor(c));
      return f;
    }
  }
}
```

```
delta D_ErrorChecking {
  modifies class Editor {
    modifies Font getFont(int c) {
      Font f = D_Editor.original(c);
      f.setUnderlined(
         getModel().isError(c));
      return f;
    } }
}
```

Finally, we add the following line to the ABS product line configuration:

delta d_f **when** f **after** d_s;

where d_s is the delta implementing the superfeature of f. If f has no superfeature, the `after` clause may be omitted. Our example requires the following product line configuration:

```
productline Editor {
  features Editor, Printing, SyntaxHighlighting, ErrorChecking;
  delta D_Editor when Editor;
  delta D_Printing when Printing after D_Editor;
  delta D_SyntaxHighlighting when SyntaxHighlighting after D_Editor;
  delta D_ErrorChecking when ErrorChecking after D_Editor; }
```

Interaction to Implement. At the feature modeling and specification level, two features f and g may be independently realizable, but require extra

functionality when both are selected. This behavior is not implemented by the feature deltas, so a new delta needs to be created. In our example, this is the case for the features Printing and Syntax Highlighting. When printing, we would like the syntax highlighting colors to be used.

Implement Interaction. The new delta $d_{f,g}$ must implement the required interaction without breaking the features f and g or their superfeatures. It may change anything introduced by feature deltas d_f and d_g. When overwriting methods, it may also access the original methods using the syntax d_f`.original()` and d_g`.original()`. In our example:

```
delta D_Printing_SyntaxHighlighting {
  modifies class Editor {
    modifies Unit print() {
      // print as before, but use colors of D_SyntaxHighlighting.font(c)
    } } }
```

Then we add the following to the ABS product line specification:

```
delta d_{f,g} when f && g after d_f, d_g;
```

In our example:

```
delta D_Printing_SyntaxHighlighting when Printing && SyntaxHighlighting
    after D_Printing, D_SyntaxHighlighting;
```

This may be generalized to interaction between more than two features.

Conflict to Resolve? By adding delta d_f, deltas $d_{f,g}$ (for different g) and conflict resolving deltas introduced earlier in the current iteration, we may have introduced an implementation conflict: two deltas d_1, d_2 that are independent, but modify the same method in a different way. In our example, this is the case for `D_SyntaxHighlighting` and `D_ErrorChecking`, as they both modify the `font` method in a different way, and are not ordered in the product line configuration. For each such conflict, we write a delta to resolve it.

Resolve Conflict. The resolving delta $d_{1,2}$ must overwrite the methods causing the conflict, while not breaking the features implemented by d_1, d_2, or their superfeatures. Typically, $d_{1,2}$ invokes d_1`.original()` and d_2`.original()` to combine the functionality of the conflicting deltas. In our example:

```
delta D_SyntaxHighlighting_ErrorChecking {
  modifies class Editor {
    modifies Font getFont(int c) {
      Font result = D_Editor.original(c);
      result.setColor(D_SyntaxHighlighting.original(c).getColor());
      result.setUnderlined(D_ErrorChecking.original(c).getUnderlined());
      return result;
    } } }
```

We add the following to the ABS product line specification:

```
delta d_{1,2} when (λ(d_1)) && (λ(d_2)) after d_1, d_2;
```

where $\lambda(d)$ is the `when` clause of delta d. In our example:

```
delta D_SyntaxHighlighting_ErrorChecking when (SyntaxHighlighting) &&
   (ErrorChecking) after D_SyntaxHighlighting, D_ErrorChecking;
```

3.3 Discussion

The workflow has some useful properties, which we briefly explain. Any feature, as well as any conflict resolution and feature interaction, can be developed independently of others that are conceptually unrelated to it. For example, all feature deltas in our example could be developed at the same time and in isolation. As could the interaction implementing delta and the conflict resolving delta.

Then there are various properties that are guaranteed in the product lines resulting from this workflow. There is a minimum of code duplication. Every delta implements some specific functionality and every product that needs that functionality employs the same delta to use it. And when two features are conceptually unrelated, two unordered deltas are developed for them. This means that two unrelated features cannot unintentionally and silently overwrite each others' code. In other words, there is no *overspecification*.

Furthermore, product lines will be globally unambiguous, meaning that for every valid feature configuration, there is a uniquely generated product. Lastly, if local constraints are met, it is guaranteed that at the end of the workflow, all necessary features and feature combinations have in fact been implemented (this is because deltas in ABS satisfy the *non-interference property* [16]). The product line implementation is *complete*.

4 Deployment Modeling

The functional correctness of a product largely depends on its high-level behavioral specification, independent of the platform on which the resulting code will be deployed. However, different deployment platforms may be envisaged for different products in a software product line, and the choice of deployment platform for a specific product may hugely influence its quality of service. For example, limitations in the processing capacity of the CPU of a cell phone may restrict the features that can be selected, and the capacity of a server may influence the response time for a service for peaks in the client traffic. In this section, we give an overview of how deployment concerns are captured in ABS models.

Modeling Timed Behavior in ABS. *Real-Time ABS* [4] is an extension of ABS in which the timed behavior of ABS models may be captured. An untimed ABS model is a model in Real-Time ABS in which execution takes zero time;

thus, standard statements in ABS are assumed to execute in zero time. Timing aspects may be added incrementally to an untimed behavioral model. Our approach extends ABS with a combination of *explicit* and *implicit* time models. In the explicit approach, the modeler specifies the passage of time in terms of duration statements with best and worst-case time. These statements are inserted into the model, and capture the duration of computations which do not depend on the deployment architecture. This is the standard approach to modeling timed behavior, known from, e.g., timed automata in UPPAAL [23]. In the implicit approach, the actual passage of time is measured during execution and may depend on the capacity and load of the server where a computation occurs.

Real-Time ABS introduces two new data types into the functional sublanguage of ABS: Time, which has the constructor Time(r), and Duration which has the constructors InfDuration and Duration(r), where r is a value of the type Rat of rational numbers. Let f be a function defined in the functional sublanguage of ABS, which recurses through some data structure x of type T, and let size be a measure of the size of this data structure. Consider a method m which takes as input such a value x and returns the result of applying f to x. Let us assume that the time needed for this computation depends on the size of x; e.g., the time needed for the computation will be between a duration 0.5*size(x) and a duration 4*size(x). We can specify an interface I which provides the method m and a class C which implements this method, including the duration of its computation using the *explicit* time model, as follows:

```
interface I { Int m(T x) }
class C implements I {
  Int m (T x){ duration(0.5*size(x), 4*size(x)); return f(x);
  } }
```

The object-oriented perspective on timed behavior is captured in terms of *deadlines* to method calls. Every method activation in Real-Time ABS has an associated deadline, which decrements with the passage of time. This deadline can be accessed inside the method body using the expression **deadline**(). Deadlines are *soft*; i.e., **deadline**() may become negative but this does not in itself stop the execution of the method. By default the deadline associated with a method activation is infinite, so in an untimed model deadlines will never be missed. Other deadlines may be introduced by means of call-site *annotations*.

Let o be an object of a class implementing method m. We consider a method n which invokes m on o, and let scale(d,r) be a scaling function which multiplies a duration d by a rational number r. The method n specifies a deadline for this call, given as an annotation and expressed in terms of its own deadline (if its own deadline is InfDuration, then the deadline to m will also be unlimited). Method n may be defined as follows:

```
Int n (T x){ [Deadline: scale(deadline(),0.9)] return o.m(x); }
```

In the *implicit* approach to modeling time in ABS, time is not specified directly in terms of durations, but rather *observed* on the model. This is done by comparing clock values from a global clock, which can be read by an expression **now**() of type Time. We specify an interface J with a method timer(x) which, given a value of type T, returns a value of type Duration, and implement timer(x) in a class D such that it measures the time it takes to call the method m above:

```
interface J { Duration timer (T x) }
class D implements J (I o) {
  Duration timer (T x){ Time start; Int y;
    start = now(); y=o.m(x); return timeDifference(now(),start);
  } }
```

With the implicit time model, no assumptions about specific durations are involved. The duration depends on how quickly the method call is effectuated by the object o. The duration is observed by comparing the time before and after making the call. As a consequence, the duration needed to execute a statement with the implicit time model depends on the *capacity* of the specific deployment model and on *synchronization* with (slower) objects.

Deployment Components. A *deployment component* in Real-Time ABS captures the execution capacity associated with a number of COGs. Deployment components are first-class citizens in Real-Time ABS, and specify an amount of resources shared by their allocated objects. Deployment components may be dynamically created depending on the control flow of the ABS model or statically created in the main block of the model. We assume a deployment component environment with unlimited resources, to which the root object of a model is allocated. When COGs are created, they are by default allocated to the same deployment component as their creator, but they may also be allocated to a different deployment component. Thus, a model without explicit deployment components runs in environment, which places no restrictions on the execution capacity of the model. A model may be extended with other deployment components with different processing capacities.

Given interfaces I, J and classes C, D as defined above, we can specify a deployment architecture, where two C objects are put on different deployment components Server1 and Server2, and interact with the D objects on the same deployment component Client. Deployment components have the type DC and are instances of the class DeploymentComponent, taking as parameters a name, given as a string, and a set of restrictions on resources. Here we focus on processing capacity, which is specified by the constructor CPUCapacity(r), where r represents the amount of available abstract processing resources between observable points in time. Below, we create three deployment components Server1, Server2, and Server3, with processing capacities 50, 100, and unlimited (i.e., Server3 has no resource restrictions). The local variables mainserver, backupserver, and clientserver refer to these deployment components. Objects are explicitly allocated to servers via annotations. The keyword **cog** indicates the creation of a new COG.

```
{ // The main block initializes a static deployment architecture:
  DC mainserver = new DeploymentComponent("Server1", set[CPUCapacity(50)]);
  DC backupserver = new DeploymentComponent("Server2", set[CPUCapacity(100)]);
  DC clientserver = new DeploymentComponent("Server3", EmptySet);
  [DC: mainserver] I object1 = new cog C;
  [DC: backupserver] I object2 = new cog C;
  [DC: clientserver] J client1 = new cog D(object1);
  [DC: clientserver] J client2 = new cog D(object2);
}
```

Resource Costs. The resource capacity of a deployment component determines how much computation may occur in the objects allocated to that deployment component. Objects allocated to the deployment component compete for the shared resources to execute, and they may execute until the deployment component runs out of resources or they are otherwise blocked. For the case of CPU resources, the resources of the deployment component define its capacity between observable (discrete) points in time, after which the resources are renewed.

The cost of executing a statement in the ABS model is determined by a default value which can be set as a compiler option (e.g., defaultcost=10). However, the default cost does not discriminate between the statements. If e is a complex expression, then the statement x=e should have a significantly higher cost than skip. For this reason, more fine-grained costs can be inserted into the model via annotations. For example, the exact cost of computing function f defined on p. 118 may be given as a function g depending on the size of the input x. Consequently, in the context of deployment components, we can specify a resource-sensitive re-implementation of interface I without predefined duration in class C2 as follows:

```
class C2 implements I {
  Int m (T x){ [Cost: g(size(x))] return f(x);
  } }
```

It is the responsibility of the modeler to specify the execution costs in the model. A behavioral model with default costs may be gradually refined to provide more realistic resource-sensitive behavior. For the computation of the cost function g in our example above, the modeler may be assisted by the COSTABS tool [1], which computes a worst-case approximation of the cost function for f in terms of the input value x based on static analysis techniques, given the ABS definition of the expression f. However, the modeler may also want to capture resource consumption at a more abstract level during the early stages of the system design, for example to make resource limitations explicit before a further refinement of a model. Therefore, cost annotations may be used by the modeler to abstractly represent some computation which remains to be fully specified. For example, the class C3 below may represent a draft version of our method m which specifies the worst-case cost of the computation even before the function f has been defined:

```
class C3 implements I {
  Int m (T x){ [Cost: size(x)*size(x)] return 0;
  } }
```

Costs need not depend merely on data values, but may also reflect overhead in general, as captured by expressions in ABS; e.g., a cost expression can be a constant value or depend on the current load of the deployment component on which the computation occurs.

Dynamic Deployment Architectures. The example presented in this section concentrates on giving simple intuitions for the modeling of deployment architectures in ABS in terms of a static deployment scenario. A full presentation of this work, including the syntax and formal semantics of such deployment architectures, is given in [13, 20]. Obviously, the approach may be extended to support the modeling of load-balancing strategies. We have considered two such extensions, based on adding an expression `load(n)` which returns the average load of the current deployment component over the last `n` time intervals. First, by including resources as first-class citizens of ABS and allowing (virtual) resources to be reallocated between deployment components [19]. Second, by allowing objects to be marshaled and reallocated between deployment components [21]. Furthermore, we have studied the application of the deployment component framework to memory resources and its integration with COSTABS in [2].

5 The ABS Component Model

Components are an intuitive tool to achieve unplanned dynamic reconfiguration. In a component system, an application is structured into several distinct pieces called *component*. Each of these components has dependencies towards functionalities located in other components; such dependencies are collected into the *output ports*. The component itself, however, offers functionalities to the other components, and these are collected into the *input ports*. Communication from an output port to an input port is possible when a *binding* between the two ports exists. Dynamic reconfiguration in such a system is then achieved by adding and removing components and by re-binding. Hence, updates and modifications acting on applications are possible without stopping them.

5.1 Related Work

While the idea of component is simple, bringing it into a concrete programming language is not easy. The informal description of components talks about the structure of a system, and how this structure can change at runtime, but does not mention program execution. As a matter of fact, many implementations of components do *not* merge into one coherent model (i) the execution of the program, generally implemented using a classic object-oriented language like Java

or C++, and (ii) the component structure, generally described in an annex Architecture Description Language (ADL). This approach makes it simple to add components to an existing language, however, unplanned dynamic reconfiguration becomes hard, as it is difficult to express modifications of the component structure using objects (as these are just supposed to describe the execution of the programs). For instance, models like Click [29] do not allow runtime modifications while OSGi [30] only allows the addition of new classes and objects: component deletion or binding modification are not supported. In this respect, a more flexible model is Fractal [5], which reifies components and ports into objects. Using an API, in Fractal it is possible to modify bindings at runtime and to add new components; still, it is difficult for the programmer to ensure that reconfiguration will not cause state inconsistencies.

Formal approaches to component models have been studied e.g., [6,25–28,34]. These models have the advantage of having a precise semantics, which clearly defines what is a component, a port and a binding (when such a construct is included). This helps to understand how dynamic reconfiguration can be implemented and how it interacts with the normal execution of a program. In particular, Oz/K [27] and COMP [26] propose a way to integrate in a unified model both components and objects. Oz/K, however, has a complex communication pattern and deals with adaptation via the use of *passivation*, which is a tricky operator [24] and in the current state of the art breaks most techniques for behavioral analysis. In contrast, COMP offers support for dynamic reconfiguration, but its integration into the semantics of ABS appears complex.

5.2 Our Approach

Most component models have a notion of component that is distinct from the objects used to represent the data and the execution of programs. Such languages are structured in two layers, one using objects for the main execution of the program, one using components for dynamic reconfiguration. Even though such a separation seems natural, it makes the integration of requests for reconfiguration into the program's workflow difficult. In contrast, in our approach we went for a uniform description of objects and components; i.e., we enhance objects and COGs—the core ingredients of ABS—with the core elements of components (ports, bindings, consistency, and hierarchy) to enable dynamic reconfiguration.

We achieved this by exploiting the similarities between objects (and object groups) and components. Most importantly, the methods of an object closely resemble the input ports of a component. In contrast, objects do not have explicit output ports, but the dependencies of an object can be stored in internal fields. Thus, rebinding an output port corresponds to the assignment of a new value to the field. Standard objects, however, cannot ensure consistency of the rebinding. Indeed, suppose we wished to treat certain object fields as output ports: we could add methods to the object for their rebinding; but it would be difficult, in presence of concurrency, to ensure that a call to one of these methods does not harm ongoing computations. For instance, if we need to update a field (like the driver of a printer), then we would first want to wait for the termination

of all current executions referring to that field (e.g., printing jobs). COGs (object groups) in ABS offer a mechanism to avoid race conditions at the level of methods, by ensuring that there is at most one task running in a COG. But this mechanism is not sufficient to deal with rebinding where we may need to wait for *several* methods to finish before performing it. Another difference between object and component models is that the latter talk about *locations*. Locations structure a system, possibly hierarchically, and can be used to express dynamic addition or removal of code, as well as distributed computation.

To ensure the consistent modifications of bindings and the possibility to ship new pieces of code at runtime, we add four elements to the ABS core language:

1. A notion of output port distinct from an object's fields. The former (identified with the keyword **port**) represent the object's dependencies and can be modified only when the object is in a *safe* state; the latter constitute the inner state of an object and can be modified with ordinary assignments.
2. The possibility to annotate methods with the keyword **critical**: this specifies that the object, while this method is executing, is not in a safe state.
3. A new primitive to wait for an object to be in a safe state. Thus, it becomes possible to wait for all executions using a given port to finish, before rebinding the port to a new object.
4. We add locations. Our semantics structures an ABS model into a tree of locations that can contain object groups, and that can move within the hierarchy. Using locations, it is possible to model the addition of new pieces of code to a program at runtime. Moreover, it is also possible to model distribution (each top-level location being a different computer) and code mobility (by moving a sub-location from a computer to another one).

The resulting component language remains close to the underlying ABS language and, in fact, is a conservative extension of ABS (i.e., a core ABS model is valid in our language and its semantics is unchanged). As shown in the following example, introducing the new primitives into a given ABS model is simple. In contrast to previous component models, our language does not strongly separate objects and components. Three major features of the informal notion of component—ports, consistency, and location—are represented in the language as follows: (i) output ports are taken care of at the level of our enhanced objects; (ii) consistency is taken care of at the level of COGs; (iii) information about locations is added explicitly.

5.3 Example

We illustrate our approach with an example inspired from the Virtual Office case study of the HATS project [10]. This case study supposes an open environment with resources like computers, projectors or printers that are used to build different workflows. Assume we want to define a workflow that takes a document (a resource modeled with the class Document), modifies it using another resource (modeled with the class Operator) and then sends it to a printer

(modeled with the class Printer). We suppose that the protocol used by Operators is complicated, so we isolate it into a dedicated class. Finally, we want to be able to change protocol at runtime, without disrupting the execution of previous instances of the workflow. Such a workflow can be defined as follows:

```
class OperatorFrontEnd(Operator op) {
 port Operator _op;

 critical Document modify(Document doc) { ... }

 { rebind _op = op; }
}

class WFController(Document doc, Operator op, Printer p) {
 port Document _doc;
 port Printer _p;
 OperatorFrontEnd _opfe;

 critical void newInstanceWF() { ... }

 void changeOperator(Operator op) {
  await(||this,_opfe||);
  rebind _opfe._op = op;
 }

 {
  rebind _doc = doc;
  rebind _p = p;
  _opfe = new OperatorFrontEnd(op);
 } }
```

We have two classes: the class OperatorFrontEnd implements the protocol in the method modify(doc); the class WFController encodes the workflow. The elements **_op**, **_doc** and **_p** are *ports* (annotated as **port**) and represent dependencies to external resources. It is only possible to modify their value using the construct **rebind**, which checks whether the object is in a safe state (no critical method in execution) before modifying the port. Moreover, methods modify(doc) and newInstanceWF() make use of these ports in their code, and are thus annotated as **critical** as it would be dangerous to rebind ports during their execution.

The key operations of our component model are shown in the two lines of code in the body of the method changeOperator(op). First is the **await** statement, which waits for objects **this** and **_opfe** to be in a safe state. By construction, these objects are in a safe state when there are no running instances of the workflow: it is then safe to modify the ports. Second is the **rebind** statement: it will succeed, because the concurrency model of COGs ensures that no workflow instance can be spawned between the end of the **await** and the end of the method. Moreover, the second line shows that it is possible to rebind a port of *another* object, provided that this object is in the same COG as the one doing the rebinding.

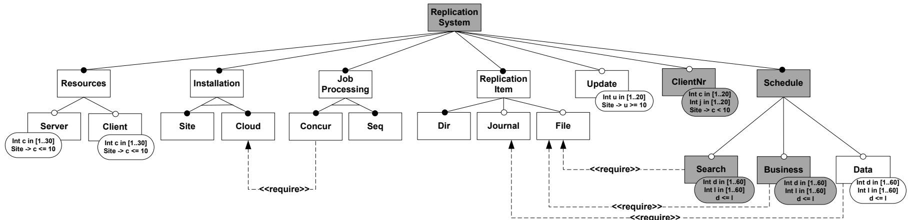


Fig. 4. Feature diagram of the replication system

6 An Industrial Case Study

The Fredhopper Access Server (FAS) is a distributed, concurrent OO system that provides search and merchandising services to e-Commerce companies. Briefly, FAS provides to its clients structured search capabilities within the client's data. FAS consists of a set of live and staging environments. A live environment processes queries from client web applications via web services. FAS aims at providing a constant query capacity to client-side web applications. A staging environment is responsible for receiving data updates in XML format, indexing the XML, and distributing the resulting indices across all live environments according to the replication protocol. The replication protocol is implemented by the *Replication System*. The replication system consists of a SyncServer at the staging environment and one SyncClient for each live environment. The SyncServer determines the schedule of replication, as well as their contents, while SyncClient receives data and configuration updates.

Modeling Variability. There are several variants of the Replication System and we express them as features. Fig. 4 shows the feature diagram of the replication system. For brevity, we consider only features *ReplicationSystem*, *ClientNr*, *Schedule*, *Search*, and *Business*. These are shaded in the feature diagram; full treatment of the complete feature diagram can be found in the HATS project report [15]. We list the μTVL model that describes these features.

```
root ReplicationSystem { group allof {
  opt ClientNr { Int c in [1 .. 20]; Int j in [1 .. 20]; },
  Schedule { group allof {
    opt Search { Int d in [1 .. 60]; Int l in [1 .. 60]; d <= l; },
    opt Business { Int d in [1 .. 60]; Int l in [1 .. 60]; d <= l; }
  } } } }
```

The replication system has the optional feature ClientNr for specifying the number of SyncClients participating in the replication protocol. It has the mandatory feature Schedule for specifying replication schedules. Replication schedules dictate when and where the replication system should monitor for changes in the staging environment to be replicated to the live environments. A replication system may offer one or both of Search and Business features. The feature Search

specifies the interval in which the replication system replicates the changes from the search index. The search index is the underlying data structure for providing search capability on customers' product items. The feature Business specifies the intervals for replicating business configuration. The business configuration defines the presentation of search results, such as sorting and promotions.

We employ the delta modeling workflow (Sect. 3) to construct an ABS model of the replication system. We start with an empty product line and define the core product as `class Main {} { new Main(); }`. Following the delta modeling workflow, we begin with the base feature ReplicationSystem. We model this feature by the delta `SystemDelta`:

```
delta SystemDelta {
  modifies class Main {
    adds Unit run() {
      List<Schedule> ss = this.getSchedules();
      Set<ClientId> cs = this.getCids();
      Int maxJobs = this.getMaxJobs();
      Int updates = this.getUpdateInterval();
      new ReplicationSystem(updates, ss, maxJobs, cs);
    } } }
```

The `run()` method creates a ReplicationSystem according to default setup. In addition, `SystemDelta` adds the necessary type definitions, such as data types, type synonyms, and core ABS classes and interfaces that model the underlying file system, the SyncClient and the SyncServer. Next to consider is the optional feature ClientNr, implemented by `ClientNrDelta`:

```
delta ClientNrDelta(Int c, Int j) {
  modifies class Main {
    modifies Set<ClientId> getCids() {
      Int s = c; Set<Int> cs = EmptySet;
      while (s > 0) { cs = Insert(s,cs); s = s-1; } return cs; }
    modifies Int getMaxJobs() { return j; }
  } }
```

The delta modifies `getCids()` and `getMaxJobs()` of class `Main` such that the replication system has synchronisation clients and a maximum number of replication jobs per client. Now we consider the mandatory feature Schedule, implemented by `ScheduleDelta`:

```
delta ScheduleDelta {
  modifies class Main {
    adds List<Pair<String,List<Item>>> searchItems = ... ;
    adds List<Pair<String,List<Item>>> businessItems = ... ;
    adds List<Schedule> getSchedules() {
      Map<String,Pair<Int,Deadline>> m = this.getScheduleMap();
      return itemMapToSchedule(Nil, m, concatenates(list[searchItems])); }
  } }
```

This delta adds methods and fields to `Main` to model various schedule information such as the types of schedules and their possible file locations from which changes are replicated. The next feature we consider is the optional feature Search, implemented by `SearchDelta`:

```
delta SearchDelta(Int d, Int l) {
  modifies class Main {
    modifies Map<String,Pair<Int,Deadline>> getScheduleMap() {
      return put(ScheduleDelta.original(), "Search", Pair(d, Duration(l))); }
  } }
```

This modifies method `getScheduleMap()` to set the interval between replicating the search index directory and the deadline for each such replication job as specified by feature Search. Since replicating the search index directory is the default schedule as defined by feature Schedule, this delta only modifies the specification of the schedule. Finally, we consider feature Business, implemented by `BusinessDelta`:

```
delta BusinessDelta(Int d, Int f) {
 modifies class Main {
  modifies Map<String,Pair<Int,Deadline>> getScheduleMap() {
   return put(ScheduleDelta.original(), "Business rules", Pair(d, Duration(l)));
  }
  modifies List<Schedule> getSchedules() { ... }
  } }
```

Similar to `SearchDelta`, this delta modifies method `getScheduleMap()` to set the interval between replicating a set of file locations and the deadline for each such replication job. In addition, it modifies `getSchedules()` to add schedules for business configuration to the replication system, the details of which we omit. We notice that `BusinessDelta` causes a conflict with `SearchDelta`. We resolve this conflict by providing the resolving delta `SBDelta`:

```
delta SBDelta {
  modifies class Main {
    modifies List<Schedule> getSchedules() {
      return appendRight(SearchDelta.original(), BusinessDelta.original()); }
  } }
```

`SBDelta` resolves the conflict between `BusinessDelta` and `SearchDelta` by insisting that the returned list of schedules contains the list of search index directory replication schedules *followed by* the list of business configuration replication schedules. Note that while both deltas modify `getScheduleMap()`, the order in which the modifications are applied needs not be specified, therefore, `SBDelta` does not provide a conflict resolver for those modifications. With no further feature interaction or conflict resolution to implement in this iteration, and no further features to implement, we obtain the complete product line.

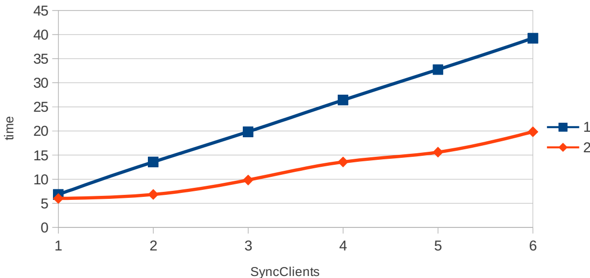


Fig. 5. Average execution time of client jobs

Resource Simulation. Using Real-Time ABS and deployment components (Sect. 4), we augment the replication system model with resource information such as processing power. We simulate the effects of processing power during execution of the replication system with the Maude backend of the ABS compiler. The following shows partial definitions of classes `ConnectionThread`, `ClientJob`.

```
class ConnectionThread {
 Unit run() {
  [Cost:size(sch)] this.start(sch); ... [Cost:length(fs)] this.register(fs); ...
  [Cost:length(fs)] this.transfer(fs); ... [Cost:size(sch)] this.finish(sch); }}
class ClientJob(Client client, ...) {
 Int total = 0;
 Unit run() { Time bt = now(); ... total = timeDifference(bt,now()); }}
```

Each `ConnectionThread` object, created by SyncServer, provides a `run()` method for interacting with a `ClientJob` object to fulfill the staging environment side of the replication protocol, while each `ClientJob` object, created by `SyncClient`, provides a `run()` method to fulfill the live environment side of the replication protocol. We provide cost annotations for specific method invocations of the `run()` method of `ConnectionThread` to describe the amount of CPU resources.[1] We inject time stamps at the beginning and the end of the `run()` method of `ClientJob` to calculate the execution time of a client job. Fig. 5 shows a graph of the average execution time (in simulated time units) of client jobs depending on the number of SyncClients and compares one vs. two CPUs. The graph shows that with a single CPU, the client job execution time increases linearly with the number of SyncClients, while with two CPUs this is no longer the case.

Unit Testing. During development of the replication system unit tests were written to validate the class methods and to detect regressions. We created the ABSUnit testing framework, based on the xUnit architecture, for writing unit tests for ABS [14]. We illustrate ABSUnit with method `processFile(id)` of

[1] Cost expressions are abstractions from concrete values obtained using the combination of real-time simulation and static cost analysis [1].

ClientJob. This method checks whether a file named id exists in the underlying database and returns its size. We define an interface ClientJobTest as the type of the test suite. It defines a test method test() and data points getData():

```
type Data = Map<Fn,Maybe<Size>>;
[Suite] interface ClientJobTest {
  [Test] Unit test(Data ds);
  [DataPoint] Set<Data> getData(); }
```

The return value of data points serves as input to the test method. The following listing shows a part of the class TestImpl that implements the methods test() and getData() from the interface ClientJobTest.

```
interface Job { Maybe<Size> processFile(Fn id); Unit setDB(DataBase db); }
[SuiteImpl] class TestImpl implements ClientJobTest {
  Set<Data> testData = ...; ABSAssert aut = ...
  Set<Data> getData() { return testData }
  Job getCJ(DataBase db) { return null; }
  Unit test(Data ds) {
    DataBase db = new cog TestDataBase(ds); Job job = this.getCJ(db);
    Set<Fn> ids = keys(ds);
    while (hasNext(ids)) {
      Pair<Set<Fn>,Fn> nt = next(ids); Fn i = snd(nt); ids = fst(nt);
      Maybe<Size> s = job.processFile(i);
      Comparator cmp = new MComp(lookup(ds,i),s);
      aut.assertEquals(cmp); }}}
```

Method test() defines a test case on processFile(id). Class MComp provides a comparator between two Maybe<Size> values. To ensure the client job object under test is prepared for unit testing, we define a delta to remove the run() method, add a setter, add a mock implementation of the database for testing, and assign type Job to ClientJob, so we can add a mock database to the object under test. This is a *major advantage of delta modeling*: code needed only for testing is encapsulated in test deltas and does not clutter up productive code.

```
delta JobTestDelta {
  modifies class ClientJob implements Job {
    removes Unit run(); adds Unit setDB(DataBase db) { this.db = db; }}
  modifies class TestImpl {
    modifies Job getCJ(DataBase db) {
      Job cj = new ClientJob(null); cj.setDB(db); return cj; }}}
```

The ABSUnit framework comes with a test runner generator that is built into to the ABS frontend. The test runner generator takes .abs files of the system under test and returns an .abs file defining a main block that executes the test cases concurrently. Here is the test runner for test interface ClientJobTest:

```
{ Set<Fut<Unit>> fs = EmptySet; Fut<Unit> f;
  ClientJobTest gd = new TestImpl(); Set<Data> ds = gd.getData();
  while (hasNext(ds)) {
   Pair<Set<Data>,Data> nt = next(ds); Data d = snd(nt); ds = fst(nt);
   ClientJobTest gd = new cog TestImpl(); f = gd!test(d); fs = Insert(f,fs); }
  Pair<Set<Fut<Unit>>,Fut<Unit>> n = Pair(EmptySet,f);
  while (hasNext(fs)) { n = next(fs); f = snd(n); fs = fst(n); f.get; }}
```

7 Conclusion

We gave an overview over the solutions to architectural issues provided by the ABS language developed in the EU FP7 project HATS. In contrast to many other behavioral modeling formalisms, ABS provides first-class support for feature models *and* connects them to implementations by a variant of feature-oriented programming called *delta modeling*. This allows to formally define a systematic *delta modeling workflow* for a feature-driven modeling process, which integrates very well with standard quality assurance techniques such as unit testing where it achieves a separation of concerns. As all software is deployed inside a wider system architecture, it is crucial to model and analyze constraints coming from deployment, which in ABS is done by *deployment components*. To structure and dynamically reconfigure a system one needs a suitable notion of components. ABS components are a conservative extension of ABS with a formal semantics. A case study, which is briefly reported in Sect. 6, demonstrates that the ABS approach scales to industrial applications. In a future paper we will concentrate on the *tool chain* shipped with the ABS environment, which contains a wide range of analysis and code generation tools.

References

1. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: COSTABS: a cost and termination analyzer for ABS. In: Kiselyov, O., Thompson, S. (eds.) Proc. Workshop on Partial Evaluation and Program Manipulation (PEPM 2012), pp. 151–154. ACM (2012)
2. Albert, E., Genaim, S., Gómez-Zamalloa, M., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Simulating Concurrent Behaviors with Worst-Case Cost Bounds. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 353–368. Springer, Heidelberg (2011)
3. Batory, D.S., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Trans. Software Eng. 30(6) (2004)
4. Bjørk, J., de Boer, F.S., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: User-defined schedulers for real-time concurrent objects. Innovations in Systems and Software Engineering (to appear, 2012), http://dx.doi.org/10.1007/s11334-012-0184-5
5. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.-B.: The Fractal Component Model and its Support in Java. Software - Practice and Experience 36(11-12) (2006)

6. Castagna, G., Vitek, J., Nardelli, F.Z.: The Seal calculus. Inf. Comput. 201(1) (2005)
7. Clarke, D., Diakov, N., Hähnle, R., Johnsen, E.B., Schaefer, I., Schäfer, J., Schlatte, R., Wong, P.Y.H.: Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 417–457. Springer, Heidelberg (2011)
8. Clarke, D., Helvensteijn, M., Schaefer, I.: Abstract Delta Modeling. In: Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, pp. 13–22. ACM, New York (2010)
9. Clarke, D., Helvensteijn, M., Schaefer, I.: Abstract delta modeling. Accepted to Special Issue of MSCS (to appear)
10. Evaluation of Core Framework. Deliverable 5.2 of project FP7-231620 (HATS) (August 2010), `http://www.hats-project.eu`
11. Report on the Core ABS Language and Methodology: Part A. Part of Deliverable 1.1 of project FP7-231620 (HATS) (March 2010), `http://www.hats-project.eu`
12. Full ABS Modeling Framework. Deliverable 1.2 of project FP7-231620 (HATS) (March 2011), `http://www.hats-project.eu`
13. A configurable deployment architecture. Deliverable 2.1 of project FP7-231620 (HATS) (February 2012), `http://www.hats-project.eu`
14. Debugging, visualization, and test generation. Deliverable 2.3 of project FP7-231620 (HATS) (March 2012), `http://www.hats-project.eu`
15. Evaluation of Modeling. Deliverable 5.3 of project FP7-231620 (HATS) (March 2012), `http://www.hats-project.eu`
16. Helvensteijn, M.: Delta Modeling Workflow. In: Proceedings of the 6th International Workshop on Variability Modelling of Software-intensive Systems, Leipzig, Germany, January 25-27. ACM International Conference Proceedings Series. ACM (2012)
17. Helvensteijn, M., Muschevici, R., Wong, P.: Delta Modeling in Practice, a Fredhopper Case Study. In: Proceedings of the 6th International Workshop on Variability Modelling of Software-intensive Systems, Leipzig, Germany, January 25-27. ACM International Conference Proceedings Series. ACM (2012)
18. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A Core Language for Abstract Behavioral Specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
19. Johnsen, E.B., Owe, O., Schlatte, R., Tapia Tarifa, S.L.: Dynamic Resource Reallocation between Deployment Components. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 646–661. Springer, Heidelberg (2010)
20. Johnsen, E.B., Owe, O., Schlatte, R., Tapia Tarifa, S.L.: Validating Timed Models of Deployment Components with Parametric Concurrency. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 46–60. Springer, Heidelberg (2011)
21. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: A Formal Model of Object Mobility in Resource-Restricted Deployment Scenarios. In: Arbab, F., Ölveczky, P. (eds.) FACS 2011. LNCS, vol. 7253, pp. 187–204. Springer, Heidelberg (2012)
22. Kang, K., Lee, J., Donohoe, P.: Feature-Oriented Project Line Engineering. IEEE Software 19(4) (2002)
23. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer 1(1-2), 134–152 (1997)
24. Lenglet, S., Schmitt, A., Stefani, J.-B.: Howe's Method for Calculi with Passivation. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 448–462. Springer, Heidelberg (2009)

25. Levi, F., Sangiorgi, D.: Mobile safe ambients. ACM. Trans. Prog. Languages and Systems 25(1) (2003)
26. Lienhardt, M., Lanese, I., Bravetti, M., Sangiorgi, D., Zavattaro, G., Welsch, Y., Schäfer, J., Poetzsch-Heffter, A.: A Component Model for the ABS Language. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 165–183. Springer, Heidelberg (2011)
27. Lienhardt, M., Schmitt, A., Stefani, J.-B.: Oz/k: A kernel language for component-based open programming. In: GPCE 2007: Proceedings of the 6th International Conference on Generative Programming and Component Engineering, pp. 43–52. ACM, New York (2007)
28. Montesi, F., Sangiorgi, D.: A Model of Evolvable Components. In: Wirsing, M., Hofmann, M., Rauschmayer, A. (eds.) TGC 2010, LNCS, vol. 6084, pp. 153–171. Springer, Heidelberg (2010)
29. Morris, R., Kohler, E., Jannotti, J., Kaashoek, M.F.: The Click Modular Router. In: ACM Symposium on Operating Systems Principles (1999)
30. OSGi Alliance. Osgi Service Platform, Release 3 (2003)
31. Schaefer, I.: Variability Modelling for Model-Driven Development of Software Product Lines. In: Proc. of 4th Intl. Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010) (2010)
32. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-Oriented Programming of Software Product Lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 77–91. Springer, Heidelberg (2010)
33. Schaefer, I., Damiani, F.: Pure Delta-oriented Programming. In: Apel, S., Batory, D., Czarnecki, K., Heidenreich, F., Kästner, C., Nierstrasz, O. (eds.) Proc. 2nd International Workshop on Feature-Oriented Software Development (FOSD 2010), Eindhoven, The Netherlands, pp. 49–56. ACM Press (2010)
34. Schmitt, A., Stefani, J.-B.: The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In: Priami, C., Quaglia, P. (eds.) GC 2004. LNCS, vol. 3267, pp. 146–178. Springer, Heidelberg (2005)
35. Wong, P.Y.H., Diakov, N., Schaefer, I.: Modelling Adaptable Distributed Object Oriented Systems Using the HATS Approach: A Fredhopper Case Study. In: Beckert, B., Damiani, F., Gurov, D. (eds.) FoVeOOS 2011. LNCS, vol. 7421, pp. 49–66. Springer, Heidelberg (2012)

Automatic Service Categorisation through Machine Learning in Emergent Middleware

Amel Bennaceur[1], Valérie Issarny[1], Richard Johansson[4], Alessandro Moschitti[3], Romina Spalazzese[2], and Daniel Sykes[1]

[1] INRIA, Paris-Rocquencourt, France
first.last@inria.fr
[2] University of L'Aquila, Italy
romina.spalazzese@di.univaq.it
[3] University of Trento, Italy
moschitti@disi.unitn.it
[4] University of Gothenburg, Sweden
richard.johansson@svenska.gu.se

Abstract. The modern environment of mobile, pervasive, evolving services presents a great challenge to traditional solutions for enabling interoperability. Automated solutions appear to be the only way to achieve interoperability with the needed level of flexibility and scalability. While necessary, the techniques used to determine compatibility, as a precursor to interaction, come at a substantial computational cost, especially when checks are performed between systems in unrelated domains. To overcome this, we apply machine learning to extract high-level functionality information through text categorisation of a system's interface description. This categorisation allows us to restrict the scope of compatibility checks, giving an overall performance gain when conducting matchmaking between systems. We have evaluated our approach on a corpus of web service descriptions, where even with moderate categorisation accuracy, a substantial performance benefit can be found. This in turn improves the applicability of our overall approach for achieving interoperability in the CONNECT project.

1 Introduction

The modern environment of mobile, pervasive, evolving services presents a great challenge to traditional solutions for enabling interoperability. The scale of complexity and heterogeneity of such devices and services, which adhere to many different standards and platforms, greatly increases the cost and difficulty of applying manual approaches. When mobility, dynamic availability, and the potential for evolution are additionally considered, the problem becomes insurmountable. Automatic approaches, termed *emergent middleware*, can overcome interoperability issues, provided that they are furnished with sufficient and relevant information, in a precise form, about the systems that should interact. This presents two sub-problems: how best to use the given information, and

B. Beckert et al. (Eds.): FMCO 2011, LNCS 7542, pp. 133–149, 2013.
© Springer-Verlag Berlin Heidelberg 2013

how to specify, extract, or discover such information. This paper addresses one case of the latter, namely, how to extract the high-level, abstract functionality information of a system, given only its detailed syntactic interface.

This high-level functionality, which we call an *affordance*, is expressed as a semantic concept from a domain ontology. Given such information we can efficiently check whether it is reasonable to attempt to make two systems interact. For example, there is little to be gained from attempting to overcome the differences between a system whose functionality is described as "Stock" and another whose functionality is described as "Weather". On the other hand there is no guarantee that two systems with the same affordance will be able to interact. To make a final assessment of compatibility, more in-depth analyses considering the interface and conversational protocol of the two systems are necessary. Avoiding such deep, time-consuming analyses motivates our use of affordances. When the affordances do not match, the detailed analyses can be omitted, providing an overall performance benefit for solving interoperability issues at runtime.

However, the requirement for affordance information places a burden on the system designer, and it is likely that legacy systems do not provide such detail. In this paper we describe an approach based on text categorisation, a machine learning technique that is able to categorise systems that have interface descriptions into affordances, based on the terms used in the interfaces. For example, an interface including many instances of the term "ticker" is likely to have the functionality corresponding to the "Stock" affordance. The assignment of affordances is thus completely automated and the full performance benefit of affordances can be reaped.

In Sections 2 and 3 we introduce the Connect project in which our work takes place, and outline the approach taken therein for discovering matching pairs of networked systems and synthesising *mediators* that enable the systems to communicate. In Section 4 we describe how text categorisation is applied to find each system's high-level functionality, and in Section 5 we show how this benefits the Discovery and Synthesis Enablers. Section 7 concludes.

2 Background

Our work takes place within the context of the Connect project[1]. The aim of the project is to overcome interoperability issues between protocols due to their heterogeneity at various levels by using an approach that dynamically generates the necessary interoperability solution that allows the systems to interact seamlessly. Connect hence promotes as a solution the dynamic synthesis of *emergent* Connect*ors* via which systems communicate. The emergent Connectors are concrete system entities synthesised according to the behavioural semantics of protocols executed by the interacting parties at application and middleware layers. The synthesis process is based on a formal foundation for Connectors, which allows learning, reasoning about and adapting the interaction behaviour of networked systems at runtime.

[1] http://connect-forever.eu/

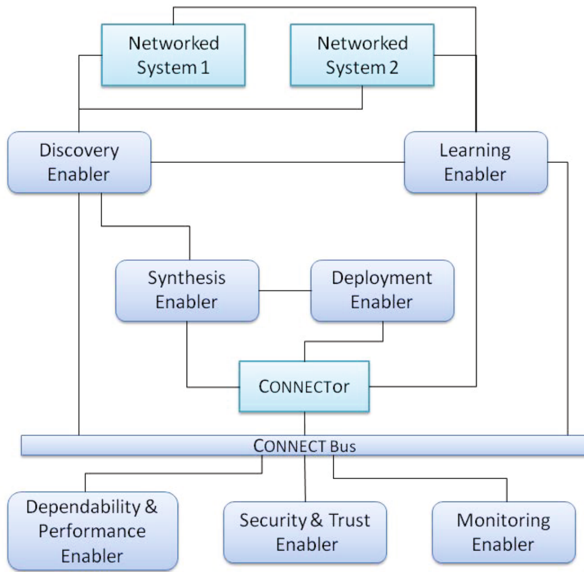


Fig. 1. Connect architecture

To reach these objectives the project undertakes interdisciplinary research, investigating the following issues and related challenges: (i) modelling and reasoning about peer system functionality; (ii) modelling and reasoning about connector behaviour; (iii) runtime synthesis of Connectors; (iv) learning peer behaviour; (v) dependability assurance; and (vi) system architecture. The architecture to realise these objectives is illustrated in Figure 1.

We call the entities that implement the mechanisms which enable the required connections *enablers*. In summary, the enablers being developed as part of the architecture are:

- Discovery Enabler: it discovers the networked systems (our generic term for services and other systems) in the environment and collects their information, including interface description and ontological description. Ontological information, in particular, is used to perform a more efficient compatibility check with other systems, i.e., to identify whether, despite possible heterogeneity, one system provides the functionality that another requires.
- Learning Enabler: it infers models of the systems' interaction behaviour, i.e., models expressing how system services can be properly invoked. This enabler leverages active automata learning algorithms.
- Synthesis Enabler: it performs a compatibility check on the system models and, if compatible, automatically synthesises a Connector that allows them to interact properly.
- Deployment Enabler: it deploys and manages the synthesised Connectors;
- Monitoring Enabler: it collects information from the Connectors, filters it, and passes it on to other requesting enablers;

– Dependability & Performance Enabler: it assesses dependability and performance properties at pre-deployment time and at runtime.
– Security and Trust Enabler: it collaborates with the Synthesis Enabler and with the Monitoring Enabler to check that possible security and trust requirements are met at runtime.

Within the described architecture, this paper focuses on the Discovery and Synthesis Enablers that benefit from the inference of high-level functionality through text categorisation.

3 Synthesising Emergent Middleware

Figure 2 outlines our overall approach to supporting emergent middleware by synthesising mediators dynamically. The key philosophy of this approach is to (i) *discover* available networked systems, (ii) *complete the descriptions* of networked systems, (iii) find *matching* pairs among them by analysing the descriptions of the networked systems, and (iv) *synthesise* mediators that allow them to interact by overcoming their incompatibilities.

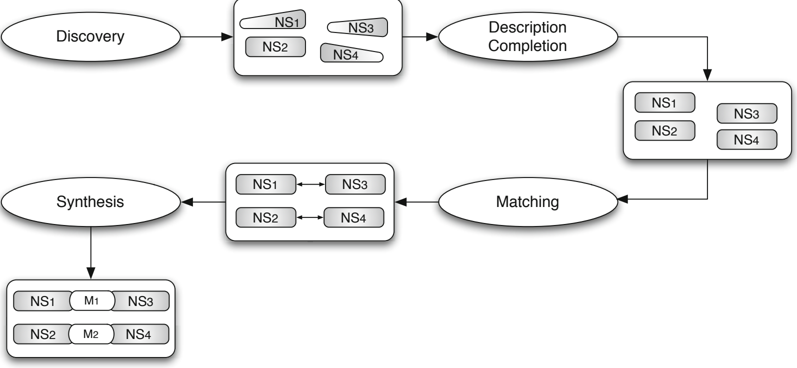


Fig. 2. Steps of creating emergent middleware

Networked systems (NSs) are discovered by the Discovery Enabler. Their descriptions may be incomplete, leading the Discovery Enabler to invoke mechanisms that can infer the missing information. To infer the interaction behaviour, the Learning Enabler is invoked, while in this paper we introduce an additional *affordance classifier* that is able to infer the system's high-level functionality.

Given two complete networked system descriptions, the next step consists of checking their *functional* compatibility, i.e., whether at high level of abstraction, the functionality required by one system can be provided by the other (see Figure 3-❶). Functional matching is performed by checking the semantic compatibility of the networked systems' affordances using ontology reasoning. When a pair of NSs have compatible functionality, we verify that they can be made

interoperable so as to achieve this functionality through *behavioural matching* (see Figure 3-❷), which is performed by analysing the behaviour of both systems. Subsequently, we synthesise the appropriate mediator which allows the two systems to communicate (see Figure 3-❸).

Ontologies play a crucial role in supporting automatic service composition [1]. They formalise the domain-specific knowledge by describing the concepts of this domain, the functions, and relations between them [2]. Ontology reasoning is particularly important for inferring the relations between concepts in open environments [1], i.e., environments that consist of many interacting systems that are developed by different vendors and are either absolutely unaware of or have only partial knowledge about the global system.

In the remainder of this section, we describe the model of networked systems that allows us to reason about their ability to interoperate. Then we describe the different steps of the matching and synthesis process.

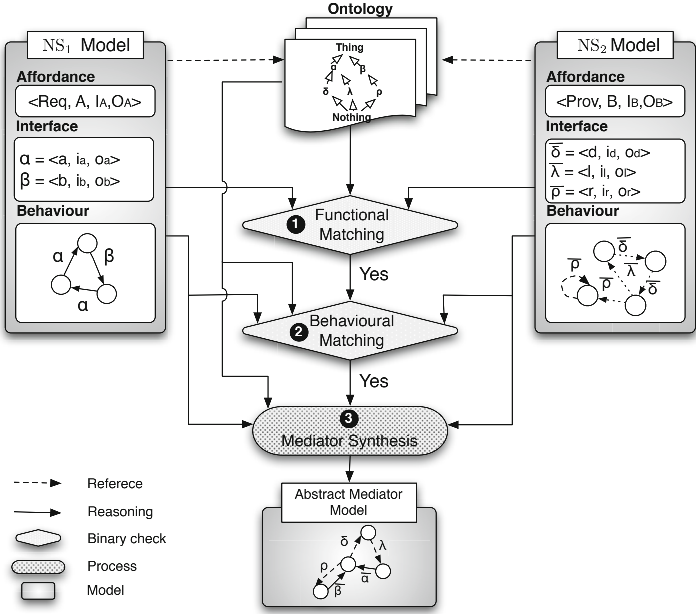


Fig. 3. Matching and synthesis

Networked System Model

A networked system requires or provides an *affordance* to which it gives access via an explicit *interface*, and which it realises using a specific *behaviour*.

The affordance specifies the high-level functionality of a system and is defined as a tuple: $\mathcal{F}$=<t, c, i, o> where (i) t stands for provided (denoted **prov**) if the system is offering this functionality or required (denoted **req**) if it is consuming

it; (ii) c gives the semantics of the functionality in terms of an ontology concept; (iii) i (resp. o) specifies the set of the high-level inputs (resp. outputs) of the functionality, which are defined as ontology concepts. All concepts belong to the same domain ontology $\mathcal{O}$ specifying the application-specific concepts and relations, i.e., $c, i, o \in \mathcal{O}$. Note that a **req** functionality produces the inputs I and consumes the corresponding outputs O. In a dual manner, a **prov** functionality consumes the inputs I and produces the corresponding outputs O. In the following we focus on the functionality concept c without considering data, overloading the term affordance where there is no ambiguity.

The interface defines the set of observable *actions* that the system requires from or provides to its execution environment, typically provided in the form of a WSDL[2] description. An *input action* $\alpha =<op,\ i,\ o>$ $(op, i, o \in \mathcal{O})$ requires an operation op for which it produces some input data i and consumes the output data o. Its dual *output action*[3] $\overline{\beta} =<\overline{op},\ i,\ o>$ uses the inputs and produces the corresponding outputs. An interface $\mathcal{I}$ is then defined as: $\mathcal{I} = \{<op_\alpha,\ i_\alpha,\ o_\alpha>\} \cup \{<\overline{op_\beta},\ i_\beta,\ o_\beta>\}$.

The system behaviour describes its interaction with its environment and defines how the actions of its interface are co-ordinated to implement a specific affordance. We build upon state-of-the-art approaches to formalise system interaction using labelled transition systems (LTS) [3].

Ontology-Based Functional Matching

Functional matching assesses whether the networked systems are functionally compatible using the following definition. A system requiring the functionality $\mathcal{F}_R$=<**req**,c_R, i_R, o_R> and a system providing the functionality $\mathcal{F}_P$=<**prov**, c_P, i_P, o_P>, are *functionally compatible*, written $\mathcal{F}_P \hookrightarrow \mathcal{F}_R$, *iff* in the associated ontology:

- c_P is a subtype of c_R,
- i_P is a supertype of i_R (contravariant), and
- o_P is a subtype of o_R (covariant).

following the Liskov substitution principle [4]. Intuitively, the $\mathcal{F}_P$ should provide at least the functionality required by $\mathcal{F}_R$ and may provide more.

Ontology-Based Behavioural Matching

Behavioural matching assesses whether the networked systems are behaviourally compatible, i.e., whether there exists an intermediary system (a mediator) through which they can safely interact. Towards this end, we first infer the correspondence between the actions of the systems' interfaces so as to generate the mappings that perform the necessary translations between semantically-compatible actions. Various mappings relations may be defined, which primarily

[2] http://www.w3.org/TR/wsdl
[3] Note the use of an overline to denote output actions.

differ according to their complexity and inversely proportional flexibility. These mappings are generated according to the mediator capabilities, which includes receiving and sending messages, delaying the delivery of messages, and reasoning about the semantics of actions in order to generate actions by transforming and composing the original ones. We use an ontology-based model checking technique to explore the various possible mappings in order to produce a correct-by-construction mediator that guarantees that the two systems can successfully interact. Model checking is used to assess system correctness and automatically verify concurrent systems by exhaustively exploring the state space, which may be very large due to state space explosion. Although many solutions have been proposed to alleviate this issue at runtime [5], behavioural matching remains substantially more costly than functional matching.

Ontology-Based Mediator Synthesis

The mediator enforces interoperation between functionally and behaviourally compatible systems despite their disparities. Mediator synthesis relies on the mappings computed during behavioural matching and refines them according to the characteristics of each networked systems and to the environment.

The specification of system functionality plays a valuable role in generating emergent middleware. It has also been acknowledged as crucial in open environments [6]. However, most legacy systems only exhibit their interface description. Hence, only partial knowledge about the system can be discovered. Given the central role of the functional matching of affordances in reducing the overall computation by acting as a filter for the subsequent behavioural matching, it is important to infer additional knowledge about the functional semantics of each networked system. Toward this goal, we use machine learning to extract the affordance of networked systems.

4 Affordance Learning and Categorisation

The problem consists in learning a *classifier* that is able to assign an affordance (specifically the functionality concept c) to a networked system automatically. The networked system has not been seen before, and its description includes an interface expressed in WSDL, but no affordance information. Note that it is not always necessary to have an absolutely correct affordance since falsely-identified matches may be caught in the subsequent detailed checks. Indeed, the pathological case with many false positives and no false negatives is equivalent to performing no affordance matching.

Since the interface is described by textual documentation, we can capitalise on the long tradition of research in *text categorisation* (TC). This studies approaches for automatically enriching text documents with semantic information (metadata). The latter is typically expressed by topic categories: thus TC proposes methods to assign documents to one or more categories. In our case, the documents to categorise are interface descriptions, and the categories correspond

to affordances. The size of the taxonomy may be small in some cases, such as a binary set, e.g., {Positive, Negative} when classifying a customer review as positive or negative [7], and larger in other cases, such as the various structured classification systems used in library science. The main tool for implementing modern systems for automatic document classification is machine learning applied to documents represented with vector space models.

In order to be able to apply standard machine learning methods for building categorisers, we need to represent the objects we want to classify by extracting informative *features*. Such features are used as indications that an object belongs to a certain category. For categorisation of documents, the standard representation of features maps every document into a vector space using the *bag-of-words* approach [8]. In this method, every word in the vocabulary is associated with a dimension of the vector space, allowing the document to be mapped into the vector space simply by computing the occurrence frequencies of each word. For example, a document consisting of the string "get Weather, get Station" could be represented as the vector $(.., 2, .., 1, .., 1, ..)$ where, e.g., 2 is the frequency of the "get" token. The bag-of-words representation is considered the standard representation underlying most document classification approaches. In contrast, attempts to incorporate more complex structural information have mostly been unsuccessful for the task of categorisation of single documents [9] although they have been successful for complex relational classification tasks [10].

However, the task of classifying interface descriptions is different from classifying raw textual documents. Indeed, the interface descriptions are *semi-structured* rather than unstructured, and the representation method clearly needs to take this fact into account, for instance, by separating the vector space representation into regions for the respective parts of the interface description. In addition to the text, various semi-structured identifiers should be included in the feature representation, e.g., the names of the method and input parameters defined by the interface. The inclusion of identifiers is important since: (i) the textual content of the identifiers is often highly informative of the functionality provided by the respective methods; and (ii) the free text documentation is not mandatory and may not always be present. In a WSDL interface, we may have tags and structures as illustrated by the text fragment in Figure 4.

It is clear that splitting the CamelCase identifier `WeatherForecastSoap` into the tokens `soap`, `weather`, and `forecast` would provide more meaningful and generalised concepts, which the learning algorithm can use as features. Indeed, to extract useful word tokens from the identifiers, we split them into pieces based on the presence of underscores or CamelCase; all tokens are then normalised to lowercase.

Once the feature representation is available, we use it to learn several classifiers, each of them specialised to recognise if the WSDL expresses some target semantic properties. The latter can also be concepts of an ontology. Consequently, our algorithm may be used to learn classifiers that automatically assign ontology concepts to actions defined in NS interfaces. Of course, the additional use of domain (but at the same time general) ontologies facilitates the learning process

```
<wsdl:portType name="WeatherForecastSoap">
  <wsdl:operation name="GetWeatherByZipCode">
    <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    Get one week weather forecast for a valid Zip Code (USA)
    </wsdl:documentation>
    <wsdl:input message="tns:GetWeatherByZipCodeSoapIn" />
    <wsdl:output message="tns:GetWeatherByZipCodeSoapOut" />
  </wsdl:operation>
  <wsdl:operation name="GetWeatherByPlaceName">
    <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    Get one week weather forecast for a place name (USA)
    </wsdl:documentation>
    <wsdl:input message="tns:GetWeatherByPlaceNameSoapIn" />
    <wsdl:output message="tns:GetWeatherByPlaceNameSoapOut" />
  </wsdl:operation>
</wsdl:portType>
<wsdl:portType name="WeatherForecastHttpGet">
  <wsdl:operation name="GetWeatherByZipCode">
    <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    Get one week weather forecast for a valid Zip Code (USA)
    </wsdl:documentation>
    <wsdl:input message="tns:GetWeatherByZipCodeHttpGetIn" />
    <wsdl:output message="tns:GetWeatherByZipCodeHttpGetOut" />
  </wsdl:operation>
  <wsdl:operation name="GetWeatherByPlaceName">
    <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    Get one week  weather forecast for a place name (USA)
    </wsdl:documentation>
    <wsdl:input message="tns:GetWeatherByPlaceNameHttpGetIn" />
    <wsdl:output message="tns:GetWeatherByPlaceNameHttpGetOut" />
  </wsdl:operation>
</wsdl:portType>
<wsdl:portType name="WeatherForecastHttpPost">
  <wsdl:operation name="GetWeatherByZipCode">
    <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    Get one week weather forecast for a valid Zip Code (USA)
    </wsdl:documentation>
    <wsdl:input message="tns:GetWeatherByZipCodeHttpPostIn" />
    <wsdl:output message="tns:GetWeatherByZipCodeHttpPostOut" />
  </wsdl:operation>
  <wsdl:operation name="GetWeatherByPlaceName">
    <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    Get one week weather forecast for a place name (USA)
    </wsdl:documentation>
    <wsdl:input message="tns:GetWeatherByPlaceNameHttpPostIn" />
    <wsdl:output message="tns:GetWeatherByPlaceNameHttpPostOut" />
  </wsdl:operation>
</wsdl:portType>
```

Fig. 4. WSDL fragment for a weather service. Our approach treats such documents as unstructured text.

Table 1. Most highly weighted features in the two-category experiment

Category	**Features**
Stock	stock, list, symbol
Weather	weather, zip, forecast, code

by providing effective features for the interface representation. In other words, WSDL, domain ontologies and any other information contribute to defining the vector representation used for training the concept classifiers.

To demonstrate the validity of the approach empirically, we experimented with automatic classification of service topics. These can be used to characterise the affordance associated with an interface (i.e., using such concepts), from which it can be inferred if two NSs are implementing compatible affordances or not.

For this purpose, we collected a set of 14 WSDL descriptions to which we manually assigned two affordance labels (i.e., categories): 8 descriptions were classified as "Stock" category of Web services, i.e., dealing with stock-markets, and 6 descriptions in the "Weather" category, i.e., weather-related services. It is clear that knowing if the offered services belong to the categories above would help to determine the affordance.

The critical aspect is to find out if such categorisation can be automatically carried out by our machine learning approach. Thus we applied rigorous statistical methods for assessing its performance. In particular, we carried out a 3-fold cross-validation over the above-mentioned dataset. To train the models, we used linear support vector machines from LibLinear software [11]. In all three experiments (three folds), the achieved precision was 100%, i.e., the classifier was always able to choose the right category for the unknown interface.

Additionally, we analysed which were the most important features of the adopted interface representation. For this purpose, we recall that the support vector learning procedure results in a *weight vector* where each dimension corresponds to the dimensions used in the feature vectors. The magnitude of these weights can be interpreted as a measure of the importance of the respective features. Table 1 shows the most highly weighted features for the two categories. As we can see, the most prominent are perfect representatives of the classes: for the "Stock" category, the algorithms has decided that `stock` is the most important feature, and similarly for the "Weather" category.

However, testing on only two categories may not provide realistic findings as many more concepts are typically involved in NS interfaces. Therefore, to evaluate our approach in a more concrete application scenario, we used a collection of WSDL documents available on the Web[4]. Note that these introduce two sources of complexities: (i) a larger number of concepts and (ii) the WSDL files do not contain natural language descriptions, which clearly facilitate semantic extraction, i.e., semantic categorisation.

We selected the 10 most frequent categories for a total of 402 documents, and we trained and evaluated the classifiers using 8-fold cross-validation. In this

[4] `http://www.andreas-hess.info/projects/annotator/ws2003.html`

Table 2. Performance by category

Category	*P*	*R*	*F*	*n*
Mathematics	0.29	0.20	0.24	23
Business	0.17	0.08	0.11	46
Communication	0.71	0.80	0.75	49
Converter	0.57	0.63	0.61	65
CountryInfo	0.64	0.83	0.72	38
Developers	0.18	0.11	0.14	46
Finder	0.55	0.59	0.57	10
Money	0.72	0.72	0.72	56
News	0.70	0.63	0.67	30
Web	0.47	0.46	0.47	39

Table 3. Most highly weighted features in the ten-category experiment

Category	**Features**
Mathematics	calculator, previous, at, value
Business	description, chart, parent, n
Communication	send, message, email, subject
Converter	to, translate, unit, my
CountryInfo	country, state, zip, postal
Developers	reverse, text, case, generate
Finder	whois, who, iwhois, results
Money	stock, amount, card, currency
News	news, quote, day, daily
Web	key, name, valid, d

experiment, the accuracy was 58%. Table 2 shows a detailed breakdown of the result. P indicates the *precision*, which is the number of documents correctly assigned to a category compared to the number that are correctly or incorrectly assigned to that category (a precision of 1 means there are no false positives). R indicates the *recall*, which is the number of documents correctly assigned to a category compared to the number that should be assigned to that category (a recall of 1 means there are no false negatives). For example, the “CountryInfo” category has a recall of 0.83, meaning that few documents of that category were falsely assigned to another. n indicates the number of documents manually assigned to each category while $F = \frac{2PR}{P+R}$, i.e., the F-measure (harmonic means between P and R).

Again, we present the most highly weighted features for each category in Table 3. As we can see, these features are highly representative of the respective categories.

In summary, in the realistic scenarios our approach decreases its effectiveness, although preserving its applicability in tasks such as automatic affordance detection. The results are promising as we achieved good accuracy using basic TC

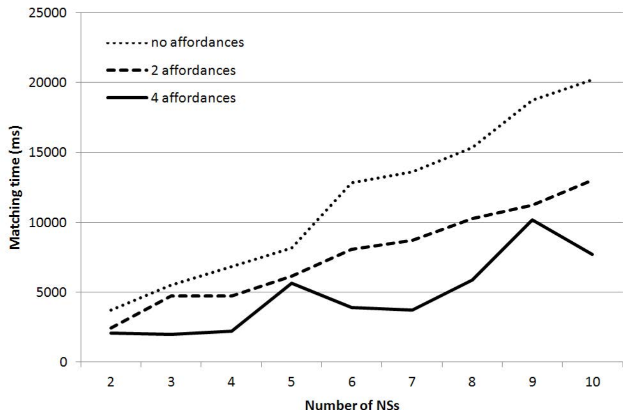


Fig. 5. Performance of matching with 0, 2, and 4 affordances

techniques to train our classifiers, although we did not use structural information and background knowledge. Also the statistical learning theory suggests greater accuracy could be achieved by increasing the size of the training data. Finally, the meaningfulness of the features selected by the classifier demonstrate that it can easily derive the best properties, alleviating the designer from the burden of manual selection.

5 Evaluation

Having shown that automatic service categorisation on the basis of interface descriptions is indeed feasible, we must now show that the affordances provided by the categorisation result in the expected benefit to discovery. The purpose of introducing affordances is to filter the number of service pairs for behavioural matching with a relatively efficient semantic check, and hence to reduce the overall time taken to conduct matchmaking when services are discovered.

After performing training offline, we integrated the trained classifier into the Discovery Enabler, which is responsible for matching pairs of networked systems. The Discovery Enabler invokes the classifier when it discovers a networked system that does not have an affordance. We then measured the time taken by the Discovery Enabler to perform matchmaking with and without the classifier.

Figure 5 shows the time taken to perform matchmaking after the sequential discovery of the given number of networked systems (up to 10). The results are averaged over ten runs. The line with the steepest average gradient shows the time taken when no affordances are used, and so no categorisation takes place. Matchmaking in this case involves performing behavioural matching for every possible pair, i.e. n^2 checks for n NSs. The other lines show the time taken when the services are automatically categorised into two and four affordances respectively. Having just two affordances reduces the number of behavioural

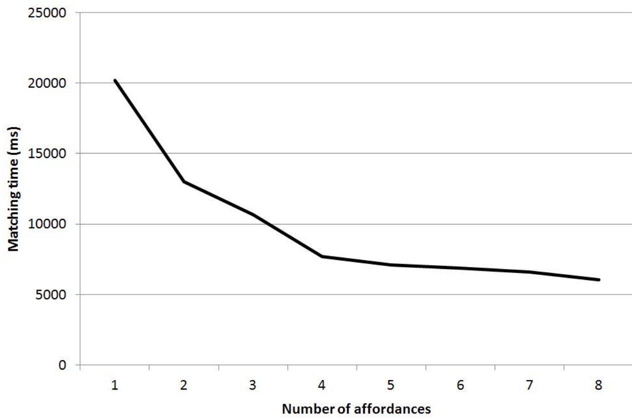


Fig. 6. Performance of matching after discovering 10 networked systems

checks to $\frac{n^2}{2}$ and adds n^2 semantic checks. In the results, we find two affordances gives a 32% reduction in the matching time, and four affordances gives a further 37% reduction.

When two or three systems have been discovered, in the case with four affordances, we do not yet expect any matches. In fact the results show an almost constant time, around 2 seconds, for matching when no matches are found. This delay represents the overhead inherent in our prototype implementation of discovery resulting from parsing WSDL and BPEL and other steps internal to discovery.

Figure 6 shows the reduction in matching time as the number of affordances increases towards the number of systems. It can be observed that the worst case time involves one affordance or none, and the best case involves as many affordances as there are networked systems (no semantic matches will be found and so no behavioural checks will be required). This suggests that the domain ontology (taxonomy) in which the affordances are defined should be as detailed as possible. Note however that increasing the number of affordances can decrease the accuracy of categorisation as features (tokens in interface descriptions) become increasingly ambiguous. This effect can be seen to an extent in the second categorisation experiment with 10 categories compared to 2 categories in the first experiment.

6 Related Work

Interoperability is a well known problem and its investigation has been done in many research contexts. For instance, in the form of supervisory control synthesis [16], discrete controller synthesis [17], component adaptors [18], protocol conversion [19,20,21], converter synthesis [22] to mention some. A work related to our mediator synthesis approach is the seminal paper by Yellin and Strom on

protocol adaptor synthesis [23] that proposes an adaptor theory to characterise and solve the interoperability problem of augmented interfaces of applications.

In more recent years increasing attention has been paid in the Web Service area to business process integration and automatic mediation, e.g., [24,25,26,27], which are related to our synthesis of mediators in some aspects. Among them, it is worth mentioning the paper [28] on behavioural adaptation because it proposes a matching approach based on heuristic algorithms to match services for the adapter generation taking into account both the interfaces and the behavioural descriptions. Moreover, the Web services community has been also investigating how to support service substitution to enable interoperability with different implementations of a service (e.g., due to evolution or provision by different vendors). While early work has focused on semi-automated, design-time approaches [26,29], latest work concentrates on automated, run-time solutions [30,31]. This latter relates to our work because of the exploitation of ontologies to reason about interface mapping and the synthesis of mediators according to such mapping.

Despite the wide range of discovery protocols that heavily rely on semantic annotations to perform service matchmaking [12,13] there are few implementations that do not assume that these services advertise their semantically-annotated descriptions. The METEOR-S Framework [14] is able to assign semantic concepts to web services by considering their WSDL descriptions but without taking into account the unstructured data potentially available within the documentation tag that can give more information about the category the web service belongs to. Instead of attaching a category concept to a web service, SAWSDL-MX2 [15] evaluates the similarity between a pair of web services based on both structured and unstructured information included in their interfaces using support vector machines. This approach is the closest to ours but is clearly not scalable especially when considering environments where services may continuously be discovered.

Moreover, these approaches only consider the functionalities of the systems to perform discovery, which may result in a false positive matching either due to the imprecision of the learning process or because the behaviour has not been considered. In our approach, the affordance matching is complemented with behavioural matching so as to match pairs of systems more accurately. Indeed, when two systems match we are able to synthesise a mediator that ensures their interoperation.

7 Conclusions

The work we have described here aims to overcome a limitation of legacy discovery mechanisms, namely that they do not provide a high-level semantic description of a system's functionality (that we call an affordance). Through the application of support vector machines for text categorisation, we have shown that the burden of categorising systems, that is, determining their high-level functional semantics, can be lifted from the engineer and performed automatically with reasonable accuracy. The cases of inaccuracy can be divided into false

positives, where two NSs have been assigned the same affordance when in fact they do not match, and false negatives, where two matching NSs are assigned different affordances and hence no attempt to connect them will be made. Minimising the number of false negatives (i.e. maximising recall) is hence critical for Connect. Greater accuracy may be achieved by finding more nuanced features, such as the structure of the document or token proximity, on which to base the categorisation.

Given such categorisation, affordance matching allows us to reduce the number of behavioural checks performed, and thus increase the performance of the matchmaking process as a whole. Our results show that the gain is relative to the number of affordances, with just two affordances providing a 32% performance increase. This performance increase benefits our overall aim in the Connect project, which is to provide solutions for interoperability at runtime, thus requiring efficient runtime mechanisms to identify compatibility and find solutions for overcoming incompatibilities.

In future work, we plan to investigate features that improve the accuracy of the categorisation, and apply categorisation in other specific areas of Connect. For example, it is desirable for each operation in an interface (as well as the interface as a whole) to give its semantics through an associated ontology concept. The approach taken in this paper may prove applicable to this problem.

Acknowledgements. This research has been supported by the EU FP7 projects: Connect – Emergent Connectors for Eternal Software Intensive Networking Systems (project number FP7 231167), EternalS – "Trustworthy Eternal Systems via Evolving Software, Data and Knowledge" (project number FP7 247758) and by the EC Project, LivingKnowledge – "Facts, Opinions and Bias" in Time (project number FP7 231126).

References

1. Blair, G.S., Bennaceur, A., Georgantas, N., Grace, P., Issarny, V., Nundloll, V., Paolucci, M.: The Role of Ontologies in Emergent Middleware: Supporting Interoperability in Complex Distributed Systems. In: Kon, F., Kermarrec, A.-M. (eds.) Middleware 2011. LNCS, vol. 7049, pp. 410–430. Springer, Heidelberg (2011)
2. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.: The Description Logic Handbook. Cambridge University Press (2003)
3. Keller, R.M.: Formal verification of parallel programs. Commun. ACM (1976)
4. Liskov, B.: Keynote address - data abstraction and hierarchy. In: Addendum to the Proceedings on Object-Oriented Programming Systems, Languages and Applications (Addendum), OOPSLA 1987, pp. 17–34. ACM, New York (1987)
5. Calinescu, R., Kikuchi, S.: Formal Methods @ Runtime. In: Calinescu, R., Jackson, E. (eds.) Monterey Workshop 2010. LNCS, vol. 6662, pp. 122–135. Springer, Heidelberg (2011)
6. Baresi, L., Di Nitto, E., Ghezzi, C.: Toward open-world software: Issue and challenges. Computer (2006)

7. Pang, B., Lee, L., Vaithyanathan, S.: Thumbs up? Sentiment classification using machine learning techniques. In: Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing, University of Pennsylvania, United States, pp. 79–86 (2002)
8. Salton, G., Wong, A., Yang, C.S.: A vector space model for automatic indexing. Technical Report TR74-218, Department of Computer Science, Cornell University, Ithaca, New York (1974)
9. Moschitti, A., Basili, R.: Complex Linguistic Features for Text Classification: A Comprehensive Study. In: McDonald, S., Tait, J.I. (eds.) ECIR 2004. LNCS, vol. 2997, pp. 181–196. Springer, Heidelberg (2004)
10. Moschitti, A.: Kernel methods, syntax and semantics for relational text categorization. In: Proceedings of ACM 17th Conference on Information and Knowledge Management, CIKM, Napa Valley, United States (2008)
11. Fan, R.E., Chang, K.W., Hsieh, C.J., Wang, X.R., Lin, C.J.: LIBLINEAR: A library for large linear classification. Journal of Machine Learning Research 9, 1871–1874 (2008)
12. Li, H., Du, X., Tian, X.: A WSMO-Based Semantic Web Services Discovery Framework in Heterogeneous Ontologies Environment. In: Zhang, Z., Siekmann, J.H. (eds.) KSEM 2007. LNCS (LNAI), vol. 4798, pp. 617–622. Springer, Heidelberg (2007)
13. Pirrò, G., Trunfio, P., Talia, D., Missier, P., Goble, C.A.: Ergot: A semantic-based system for service discovery in distributed infrastructures. In: CCGRID, pp. 263–272 (2010)
14. Oldham, N., Thomas, C., Sheth, A.P., Verma, K.: METEOR-S Web Service Annotation Framework with Machine Learning Classification. In: Cardoso, J., Sheth, A.P. (eds.) SWSWPC 2004. LNCS, vol. 3387, pp. 137–146. Springer, Heidelberg (2005)
15. Klusch, M., Kapahnke, P., Zinnikus, I.: Sawsdl-mx2: A machine-learning approach for integrating semantic web service matchmaking variants. In: ICWS, pp. 335–342 (2009)
16. Brandin, B., Wonham, W.: Supervisory control of timed discrete-event systems. IEEE Transactions on Automatic Control 39(2) (1994)
17. Ramadge, P., Wonham, W.: Supervisory control of a class of discrete event processes. SIAM J. Control and Optimization 25(1) (1987)
18. Bracciali, A., Brogi, A., Canal, C.: A formal approach to component adaptation. J. Syst. Softw. 74 (2005)
19. Calvert, K.L., Lam, S.S.: Formal methods for protocol conversion. IEEE Journal on Selected Areas in Communications 8(1), 127–142 (1990)
20. Lam, S.S.: Correction to "protocol conversion". IEEE Trans. Software Eng. 14(9), 1376 (1988)
21. Okumura, K.: A formal protocol conversion method. In: SIGCOMM, pp. 30–37 (1986)
22. Passerone, R., de Alfaro, L., Henzinger, T.A., Sangiovanni-Vincentelli, A.L.: Convertibility verification and converter synthesis: two faces of the same coin. In: Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2002, pp. 132–139 (2002)
23. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM Trans. Program. Lang. Syst. 19 (1997)
24. Cimpian, E., Mocan, A.: WSMX Process Mediation Based on Choreographies. In: Bussler, C., Haller, A. (eds.) BPM 2005. LNCS, vol. 3812, pp. 130–143. Springer, Heidelberg (2006)

25. Vaculín, R., Neruda, R., Sycara, K.: An Agent for Asymmetric Process Mediation in Open Environments. In: Kowalczyk, R., Huhns, M.N., Klusch, M., Maamar, Z., Vo, Q.B. (eds.) SOCASE 2008. LNCS, vol. 5006, pp. 104–117. Springer, Heidelberg (2008)
26. Motahari Nezhad, H.R., Benatallah, B., Martens, A., Curbera, F., Casati, F.: Semi-automated adaptation of service interactions. In: WWW 2007: Proceedings of the 16th International Conference on World Wide Web, pp. 993–1002. ACM, New York (2007)
27. Williams, S.K., Battle, S.A., Cuadrado, J.E.: Protocol Mediation for Adaptation in Semantic Web Services. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, pp. 635–649. Springer, Heidelberg (2006)
28. Motahari Nezhad, H.R., Xu, G.Y., Benatallah, B.: Protocol-aware matching of web service interfaces for adapter development. In: Proceedings of the 19th International Conference on World Wide Web, WWW 2010, pp. 731–740. ACM, New York (2010)
29. Ponnekanti, S.R., Fox, A.: Interoperability Among Independently Evolving Web Services. In: Jacobsen, H.-A. (ed.) Middleware 2004. LNCS, vol. 3231, pp. 331–351. Springer, Heidelberg (2004)
30. Denaro, G., Pezzé, M., Tosi, D.: Ensuring interoperable service-oriented systems through engineered self-healing. In: Proceedings of ESEC/FSE 2009. ACM Press (2009)
31. Cavallaro, L., Di Nitto, E., Pradella, M.: An Automatic Approach to Enable Replacement of Conversational Services. In: Baresi, L., Chi, C.-H., Suzuki, J. (eds.) ICSOC-ServiceWave 2009. LNCS, vol. 5900, pp. 159–174. Springer, Heidelberg (2009)
32. Heß, A., Kushmerick, N.: Learning to Attach Semantic Metadata to Web Services. In: Fensel, D., Sycara, K., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 258–273. Springer, Heidelberg (2003)

Towards a Model- and Learning-Based Framework for Security Anomaly Detection*

Matthias Gander, Basel Katt, Michael Felderer, and Ruth Breu

Institute of Computer Science, University of Innsbruck, Austria
{matthias.gander,basel.katt,michael.felderer,ruth.breu}@uibk.ac.at

Abstract. For critical areas, such as the health-care domain, it is common to formalize workflow, traffic-flow and access control via models. Typically security monitoring is used to firstly determine if the system corresponds to the specifications in these models and secondly to deal with threats, e.g. by detecting intrusions, via monitoring rules. The challenge of security monitoring stems mainly from two aspects. First, information in form of models needs to be integrated in the analysis part, e.g. rule creation, visualization, such that the plethora of monitored events are analyzed and represented in a meaningful manner. Second, new intrusion types are basically invisible to established monitoring techniques such as signature-based methods and supervised learning algorithms.

In this paper, we present a pluggable monitoring framework that focuses on the above two issues by linking event information and modelling specification to perform compliance detection and anomaly detection. As input the framework leverages models that define workflows, event information, as well as the underlying network infrastructure. Assuming that new intrusions manifest in anomalous behaviour which cannot be foreseen, we make use of a popular unsupervised machine-learning technique called clustering.

Keywords: Modelling, Profiling, Machine Learning, IT-Security, Runtime-Monitoring, Anomaly Detection, Clustering.

1 Introduction

Modelling provides a way to specify security requirements and is often employed in critical areas such as the health-care sector. A standardization committee in this sector leveraging such modelling techniques, is for instance the *integrating the health-care environment* (IHE) standards committee.[1] IHE gives requirements on traffic-flow, for instance restricted communication between specific nodes (e.g. hosts, routers), and access control in form of authentication/authorization protocols. On top of a system that is supposed to adhere

* This work is supported by QE LaB - Living Models for Open Systems (FFG 822740), COSEMA - funded by the Tiroler Zukunftsstiftung, and SECTISSIMO (P-20388) FWF project.

[1] http://www.ihe.net/, Accessed: January 5, 2012.

B. Beckert et al. (Eds.): FMCO 2011, LNCS 7542, pp. 150–168, 2013.
© Springer-Verlag Berlin Heidelberg 2013

to requirements like availability, confidentiality, and integrity it is necessary to apply monitoring methods. These methods determine if said requirements are met, i.e. by preventing or detecting intrusions (IPS/IDS). A very common technique for this involves signature based methods. Features, extracted from event data are compared to features in attack signatures that are provided by experts. Other approaches use machine learning algorithms that train on labeled input data.

Signature-based methods have an inherent limitation when it comes to detect new attacks. This limitation stems from the need to consult a signature database, which not always contains the latest attack patterns. In order to improve on signature-based methods, several anomaly detection approaches were proposed to detect changes in normal behaviour [1–4]. Basic techniques in anomaly-detection include statistical-, knowledge- and machine learning-based approaches. Supervised machine learning comes with two drawbacks, the first one being the dependency on labelled training instances, which is not always easy to procure. And the second is that training instances are susceptible to be trained by an attacker [5]. Leveraging the assumption that *most behaviour in the network is normal* ($\eta\%$) *and the abnormal behaviour is just a tiny part of the overall behaviour* ($1-\eta\%$) [1] it becomes possible to apply unsupervised machine learning to detect this change in behaviour.

Contribution and Structure

The goal of this paper is to provide a framework to support real-time outlier detection in security critical networks, e.g. the health-care domain, through event analysis. This framework leverages the fact that ample information is provided through modelling information, i.e. the specification of workflows, service call sequences, and the corresponding infrastructure nodes. We will call this fusion of modelling information the *IT landscape* model. Combining the specification of service events and the infrastructure itself, it is possible to provide stateful monitoring as well as anomaly detection. Each workflow is characterized as a finite state machine (FSM) to describe the steps of a concrete workflow. The transition function itself is dependent on service events, thus relating service events to workflows. Every service event is mapped to instances of infrastructure event clusters, which are learned through training. Using state machines we detect workflow attacks and through clustering of infrastructure events we are able to create normality profiles, detect outliers and raise alerts accordingly. Our contribution is the introduction of a monitoring framework that is based on the following features:

1. A tight coupling between different layers in the software stack through the use of a UML metamodel [6] as means for specifying, network infrastructure, service events and workflows.
2. An enhancement of the presentation of non-related information, e.g. an event view that lists network infrastructure events related to workflows.

3. An automatic detection of (a) workflow modifications by analysing the internal state of a workflow, and (b) anomalous activity that has its source in the infrastructure by analysing event streams.

In Section 3 the running example, our threat model and basic concepts, i.e. our layered concept and events, are introduced. This is followed by Section 4 which presents the framework. Section 2 and 5 discuss related and future work respectively.

2 Related Work

In this section we will discuss related work in the areas of modelling, workflow compliance and anomaly detection via clustering in intrusion detection.

Modelling. The key concepts for our metamodel, e.g. the three layers, inclusion of roles, node hierarchies, derives from the work of [7, 8]. Both approaches provide a model-based approach together with concepts and methods for security management. The latter work also includes the *Living Models* [9] approach where models are subjected to change over time. Other ways to model a service infrastructure exist as well, for example the *Service Oriented Architecture Modeling Language (SoaML)*[10] and the *Service Markup Language* SML [11]. SoaML is a UML profile allowing specific concepts by extending the UML standard. There are two main reasons why we did not make use of SoaML, the first one being that SoaML was especially designed for modelling services in the context of SOA. We are interested in services, yet not all concepts of SOA are of interest, hence, SoaML provides a too rich vocabulary. The second reason is that by sticking to a similar approach as [7] and [8] we hope to be able to in the future adhere to the *Living Models* concept, to react to changes in the IT landscape. Since our interest is mainly event oriented, our metamodel is built to reflect this by leveraging and adapting the idea of event-driven process chains (EPC), deeply discussed in [12], to our needs.

Compliance Monitoring. Mulo et al. [13] propose monitoring compliance of business processes in SOA via complex event processing (CEP) means. A service invocation is regarded as an event and business process activities as event-trails. These event-trails guide the creation of queries which a CEP engine uses to identify and monitor business activities. Since the business activities are rendered identifiable it is possible to monitor the flow of a business process at runtime. Hence, it is possible to detect anomalous process executions. Baresi et al. [14] and Erradi et al. [15] focus on monitoring the execution of centrally orchestrated web services compositions (specified in WS-BPEL) in order to detect, correlate and react meaningfully to incidents. Baresi et al. [14] extends WS-Policy with a language for constraints to monitor functional and non-functional requirements (weaved with the BPEL process at deployment-time). This approach focuses on monitoring very low-level security requirements such as signature algorithms

used. Erradi et al. [15] present a hybrid approach for functional and QoS monitoring combining synchronous and asynchronous monitoring techniques. The main difference to our approach is the combination of layers to detect abnormalities in the infrastructure layer. Since our metamodel however also allows the definition of event-sequences, that can be used as workflow traces, we are also able to detect deviations of workflows, similar to [13].

Anomaly Detection. Anomaly detection is well-established in the domain of intrusion detection systems [5]. The decision to choose unsupervised machine learning is shared among multiple publications [1, 16, 2, 3]. Portnoy et al. [1] uses cluster analysis successfully to detect attacks in the KDD 1999 data set.[2] Gu et al. [3] also leverage clustering techniques (successfully) to detect botnets using their tool "Botminer". Clustering as a tool, in the area of Network Intrusion Detection Systems (NIDS), is itself also subject of ongoing research. [16] introduces adaptive clustering to reduce time-based bias in dynamic networks, i.e. traffic variance over time. [2] improves clustering for NIDS by using a density-based clustering algorithm and a grid-based memtric. Both Oldmeadow et al. [16] and Leung et al. [2] evaluate their efforts on the KDD 1999 data set. Their approaches have the same basic workflow, which is similar to all anomaly-detection frameworks, feature extraction, training and detection [5]. We also make use of KDD features and common clustering algorithms (see Section 4.3). The main differences are, we do not cluster for botnet activity, there is no cross-correlation among clusters to improve detection rate, and most notably of all we relate clustering to workflow models.

To the best of our knowledge, there exists no model-driven approach doing anomaly detection based on landscape models leveraging clustering techniques. Additionally, in contrast to most existing work, we are able to pinpoint not only outlier events, but through models also locate suspicious nodes, services and workflows.

3 Basic Concepts

In this section we discuss our motivating example, present our threat model, and provide details about concepts, such as the layered approach we use, and the events we are interested in.

3.1 Motivating Example

Our framework is explained based on a running example taken from a workflow in the health-care domain. In this scenario a host (document consumer), which could be a doctor, wants to retrieve sensitive information (patient medical record) from a document repository over an XACML infrastructure. Using this

[2] `http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`, Accessed: January 5, 2012.

example we discuss the expected behaviour of a workflow, demonstrate multiple threats, and finally show how the framework can mitigate these.

The workflow itself consists of a series of steps that involve the participation of web services. Much of the traffic in the network is SOAP over HTTP but also other protocols from the ISO/OSI stack occur. Workstations and databases are grouped into zones called affinity domains and are connected via LAN. For the description we make use of the standard BPMN and workflow notions, i.e. actors and tasks [17]. We distinguish among multiple actors in our example: The *Identity Provider*, which identifies nodes and users, but also assigns roles. The *Gateways A* and *B*, that relay queries and responses to other affinity domains. The *Document Consumer*, which is the source of a patient document query, or document change request, for instance a host requesting a patient medical record (PMR). The *Document Repository*, which is the location of the patient data. Access control is offered through an XACML architecture.

A document retrieval transaction, which involves polling a patient's confidential information basically consists of two major tasks, namely authentication, and document retrieval. Below we explain the tasks in the sample workflow from Figure 1:

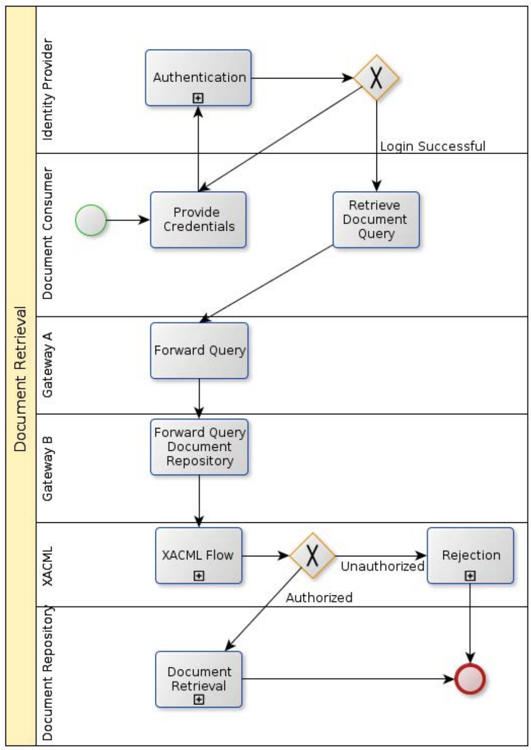


Fig. 1. Workflow for a Document Retrieval

- *Provide Credentials*: The document consumer (DC) contacts the identity provider and forwards his credentials.
- *Authentication*: The identity provider queries a database interface service to validate the credentials and assign a role to the user and answers with a SAML 2.0 assertion containing an authorization statement.
- *Retrieve Document Query*: After the document consumer receives this ticket, the actual retrieval can start. The consumer sends a retrieve document query together with his ticket to the gateway of his affinity domain.
- *Forward Query*: The gateway forwards the ticket to the corresponding gateway of the affinity domain that contains the data.
- *Forward Query Document Repository*: The receiving gateway forwards the query to the document repository where an XACML [18] flow starts, consider Figure 2. The most important assets in the architecture are the following:
  1. The *policy enforcement point* (PEP) attached to the document repository calls the PIP to deduce attributes for the document that needs to be retrieved.
  2. The *policy information point* (PIP) returns the appropriate attributes.
  3. The PEP forwards the query and resources to the PDP and awaits a decision.
  4. The *policy decision point* (PDP) acquires policies for the target from the PAP.
  5. The *policy assertion point* (PAP) returns the appropriate policies and the PDP, via rule combining algorithms, derives a decision.
  6. The PEP receives the decision and either (depending on the ruling of the PDP) allows or rejects the access.
- Now either the rejection message (*Rejection*), due to unauthorized access, or the document itself (*Document Retrieval*) is transmitted to the document consumer, concluding the workflow.

During the execution of this workflow many expected events are generated in the service area, namely SOAP calls. These expected service events are related to infrastructure events, like, database access and network traffic in form of TCP and UDP traffic. Unfortunately, also other events occur which are very similar to infrastructure events related to service events, i.e. TCP/HTTP packets which arise from normal web-browsing employees (Facebook, News), instant messaging (Skype, ICQ, ...), UDP traffic from DNS and LDAP queries. Relating a subset of all infrastructure events to expected service events is one part of our work.

3.2 Threat Model

If an attacker wished to gain confidential information a common way to do so would be infiltration, i.e. stealing credentials from a doctor or hacking into a node in the network. After an attack, it is often prudent to install backdoors, i.e. by patching or replacing code [19, 20]. But not all attacks to gain information rely on the same techniques [21, 22]. In our experience attacks manifest either in workflows, execution tampering, or a layer beneath, in the infrastructure.

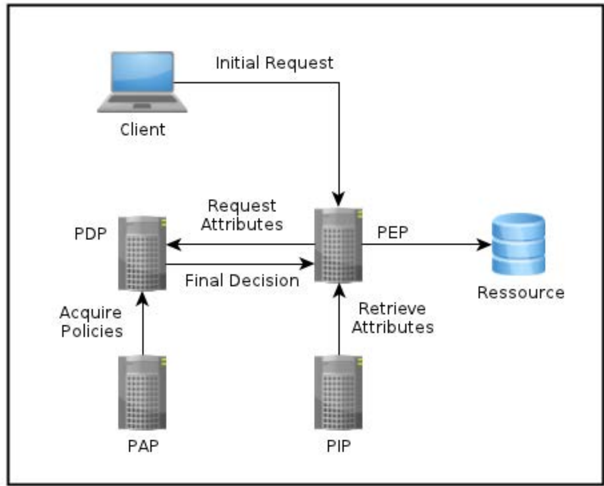


Fig. 2. A simplified XACML Data Flow model

Workflow Attacks. The attacker chooses to deliberately sabotage the service invocation sequence. Concerning our example a likely attack destination is the XACML infrastructure. Assume the PEP does not include the PDP in the access control flow, and allows anyone, or selected individuals, to access PMRs. Another example is the communication between PDP and PIP. The PDP is supposed to retrieve document attributes from the PIP to base its decision making. What if the PDP does not? Services along the way of document queries are also susceptible to attacks, for example replay-attacks or fake queries (false tickets). These attacks manifest in missing events, wrong sequences of events, sometimes too many (replay attacks), along the execution of the workflow.

Infrastructure Attacks. A multitude of potential attacks are possible in this perspective, leveraging attack vectors of web services,[3] buffer overflows in "low-level" services like, FTP, and badly configured access control, i.e. allowing SMB null sessions. Such attacks manifest in odd connection patterns, string patterns, and access patterns. The whole set of possible attacks is impossible to cover, but we assume that most attacks manifest in abnormal behaviour.

Example 1. In the following Table 1 we illustrate an anomaly that portraits an access rights manipulation in the activity *Retrieve Document Query*. The query shown in Ev_2 (from the document consumer with UID_x and requested resource object, PMR_{12}), results in an *Authorization Denied* event emitted from the PDP. Immediately after the *Authorization Denied* event, the database is accessed at the document repository for the same resource and more importantly from the

[3] `http://www.ws-attacks.org`, Accessed: January 5, 2012.

same user, shown in Ev_5. Above all, as we will show, the time of the query is very unusual for the user UID_x.

Example 2. In Table 2, after the *Retrieve Document Query* from the DC with UID_x has been sent to the DR, the DC sends TCP packets to two hosts C and D in the network. C is used for backup services, yet D in this instance represents a malicious host that collects confidential data.

Table 1. Access Rights Manipulation

Event	Layer	Source	Dest	App	Ident	Object	Time	Type
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
Ev_1	Service	DC	G	–	UID_x	PMR_{12}	2.00 am	Query
Ev_2	Service	G	DR	–	UID_x	PMR_{12}	2.00 am	Retrieve
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
Ev_3	Infrastructure	PDP	–	Postgres SQL	PDP	UID_x	2.01 am	Read
Ev_4	Service	PDP	PEP	–	UID_x	PMR_{12}	2.01 am	AuthDenied
Ev_5	Infrastructure	DR	–	Postgres SQL	UID_x	PMR_{12}	2.01 am	Update
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 2. Abnormal Communication Pattern

Event	Layer	Source	Dest	App	Ident	Object	Time	Type
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
Ev_1	Service	DC	DR	–	UID_x	PMR_{12}	7.20 am	Query
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
Ev_2	Service	DR	DC	–	UID_x	PMR_{12}	7.23 am	Retrieve
Ev_3	Infrastructure	DC	C	–	–	–	7.23 am	TCP
Ev_4	Infrastructure	DC	D	–	–	–	7.23 am	TCP
Ev_5	Infrastructure	DC	D	–	–	–	7.23 am	TCP
Ev_6	Infrastructure	DC	C	–	–	–	7.23 am	TCP
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

3.3 Layered Concept and Associated Events

Making the monitor aware of the IT landscape provides more means for detection. For instance, by linking service call sequences to workflows to detect a possible XACML flow anomaly. Splitting an enterprise model, as we do here for workflow execution, into three layers, has alread been established [7, 8]. The first

layer is the *workflow layer*, it consists of an abstract flow of activities, that determines the sequence of a workflow. Activities do not execute code themselves, this task is done via services in the network, hence, the second layer is the *service layer*. One layer below is the *infrastructure layer*, in here, nodes host services and represent the backbone of executing workflows.

Our focus lies on *service* and *infrastructure* layer events. Service events are calls from one service to another that have a type. In this work we cover a set of standard types, *authentication, authorization, query, retrieve, delete, update, create* and *forward*. Infrastructure events are split into network and database events. Network events are UDP and TCP packets, with descriptive features such as, source, destination, ports, time, among others. For database events it is essential to know all the details of a create, read, update, and delete (CRUD) operation. This means events need to contain a user id, the object, the time, the type of CRUD operation and the originating host. We do not explicitly envision the use of workflow events because it is already possible to derive information about the workflow layer by just monitoring service events.

4 Approach

In this section we present our anomaly detection framework, which consists of a *modelling component*, that enables defining UML models to represent an IT landscape (see Figure 1), a *Finite State Machine (FSM) generator* that generates FSMs for the purpose of compliance detection from workflow models, *sensors* and an *analyzer* as event sink to cluster and analyze data. Two steps are necessary for running the framework: (1) modelling the IT landscape, and (2) configuration, which involves configuring sensors on services and nodes but also white-listing of “friendly” traffic.

4.1 IT-Landscape Metamodel

Our metamodel allows a designated expert, or group of experts, to model a network infrastructure that provides the underpinning to a workflow, as shown in Figure 3. The model reuses concepts from [7, 8], for example the introduction of multiple conceptual layers to create a realistic enterprise model. In contrast to [7, 8] in our modelling approach we follow an event-driven process chain paradigm [13]. This has the benefit that we do not need to model services and events, but only model events and see the services (and the application they represent) implicitly. A workflow activity, therefore, is not modelled via services and their call-sequence but rather as a series of events.

Our model contains the layers, *Workflow, Service* and *Infrastructure*. Starting from the workflow layer, we distinguish among *WF Activity, Role*, and *Actor*. A workflow consists of two or more workflow activities which are connected via arcs (*Arc*) that are of different types *AND, OR, XOR, SEQ*. The semantics behind *SEQ* are simple, an arc between two workflow activities A and B denotes that A is followed by B. *AND, OR* and *XOR* make use of multiple events and

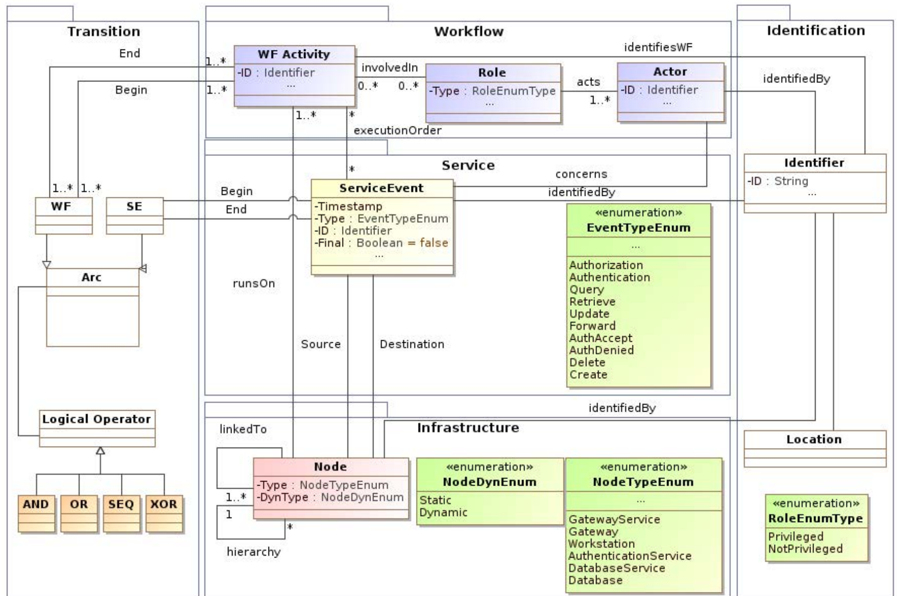


Fig. 3. The IT landscape Metamodel

relate them in semantics known from boolean operations (see Figure 4). To each activity a role is attached. A role is a set of responsibilities and obligations for a stakeholder. A set of actors is associated to roles which are uniquely identifiable, via an *Identifier*. Services are not modelled directly, but rather as *ServiceEvent* of various types (*EventTypeEnum*). Event emitters are services, on top of network nodes. Hence, among other features provided by the service event, i.e. dynamic ones like timestamps and session ids (to identify the *Actor*, we assume a source and a destination pointing to the nodes that took part in the event. This allows us to connect the service layer to the infrastructure layer. Nodes can be of various type (*NodeType*), this makes it easier to map events to their corresponding workflow activity during runtime. Each one of the many association relations denotes the element's use. *Identifier* defines the set of identifiers, i.e. all elements are connected to it via *identifiedBy*, such as nodes, service events, and actors are identified by it (via UUID and a location). Workflow activities are executed via services, the execution order is in form of service events. The real elements doing the execution are nodes from the infrastructure (*runsOn*).

Example. The workflow activity, *Authentication*, from our running example (see Section 3.1) can now be modelled, consider Figure 4. The activity is associated to five events, $\text{Ev}_{1,\ldots,5}$. The execution takes place on nodes in the infrastructure. These nodes are split into static ones, *Authentication Service, Database*, and a dynamic one, *Workstation*. The reason for dynamic and static nodes is

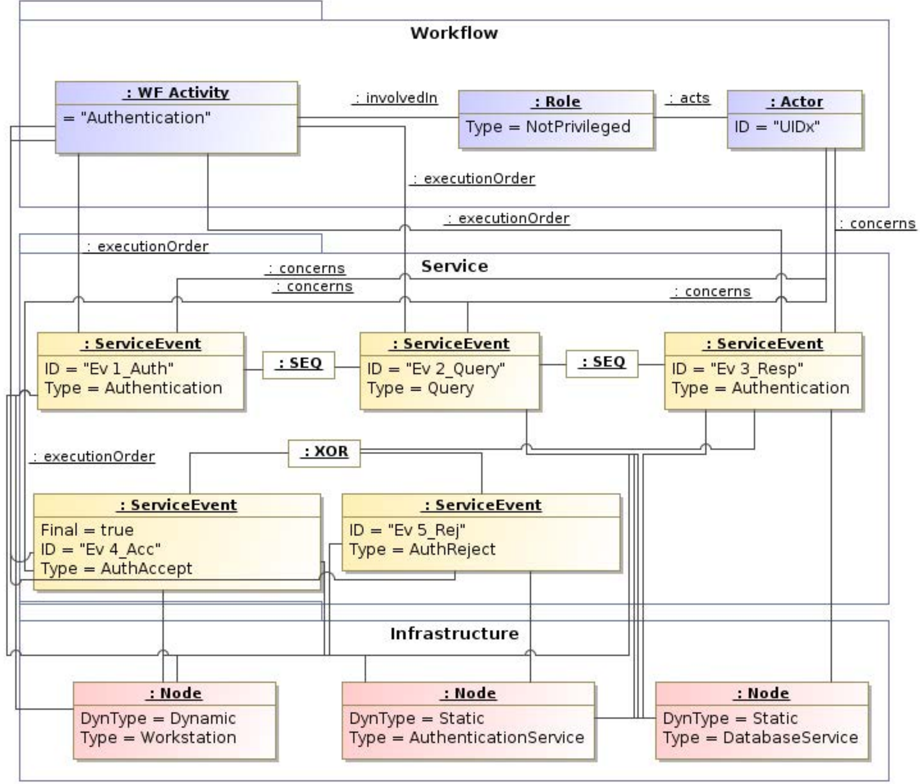


Fig. 4. Sample model for the authentication activity of our running example

that some information about nodes is dynamic and subject to frequent change on runtime, for instance nodes with IP addresses over non-static DHCP. Other information stays static for a longer period, i.e. non-mutable DNS names for Windows Domain Controllers. As a result, the dynamic classifier can be seen as wildcard. By permitting dynamic nodes in models it, therefore, becomes possible to have a dynamic composition of workflows. Within the example any node that is a *Workstation* is allowed to participate in the authentication workflow (the result will still depend on the credentials though). This distinction, thus, renders the linking of event trails to the correct workflow activity possible.

4.2 Workflow State Machines

During the runtime of the system we collect service calls. Some questions about these calls are easy to answer. For instance, *"Who was involved in it?"* or *"Which node emitted it?"*. Some other questions are trickier, i.e. mapping a series of events (*event trails* [13]) to the correct workflow activities and detect aberrations in the sequences.

To answer these questions the metamodel is used to create a workflow model through the specification of the sequence of events that are expected. This facilitates the instantiation of an FSM that represents an internal representation of each workflow. FSMs and the workflows defined via the model (see Figure 4) are very similar, still, an internal representation via FSMs has the benefit of (a) the formalisms are well-understood and (b) there are libraries that have been developed to handle large (complex) state machines efficiently.[4]

Translation. A deterministic finite state machine or acceptor deterministic finite state machine is a quintuple $(\sigma, S, s_0, \delta, F)$, where σ is the input alphabet (a finite, non-empty set of symbols), S is a finite, non-empty set of states, $s_0 \in S$ is an initial state, δ is the state-transition function: $\delta : S \times \Sigma \rightarrow S$, and $F \subseteq S$ is the set of final states. During the modelling phase events are created and associated to workflow activities. The translation from workflow model to a state machine is straight forward, the events are already modelled as a sequence. The logical operators, "AND, XOR, OR, SEQ", are translated to multiple states that can either be reached with one or the other event but not both ("XOR"). "AND" denotes that a state transition needs multiple events to reach the follow-up state, and "OR" translates to a case similar to "XOR", albeit with weaker conditions, both events are allowed to happen.

FSM Lifecycle. Each event now serves as a state transition, which makes it possible to track the behaviour of the workflow at runtime. The lifecycle of an FSM is as follows: Any transaction (e.g. authentication, document retrieval) emits events. Such an event includes, e.g. type, source, destination, and a session id identifying the actor. If an event is encountered which has a new session id, a new FSM has to be instantiated. To instantiate the appropriate FSM for the workflow we assume that the combination of the event characteristics, i.e. event type, source, destination, session id, uniquely identifies the workflow. Applying this method we can carry out basic workflow compliance detection based on service events.

Examples. The authentication activity from Figure 4 would lead to a state machine consisting of 6 states (the naming is arbitrary), $S = \{\text{Init}, 2, 3, 4, \text{Acc}, \text{Rej}\}$ where s_0 = Init, F = {Acc, Rej} and state transitions, for instance δ : (Init, Ev_1) = 2, for all states S and events, $\text{Ev}_{1...5}$.[5] Ev_1 is the event describing how a client contacts the authentication service, Ev_2 is the query from the authentication service to the database service. The database service answers in event Ev_3 and the authentication service either, grants access Ev_4 or denies to do so Ev_5. The state machine above only describes a particular activity. But by conjoining activities of our example and their FSM representations, a simple task [23], we are able to create a complete FSM for the whole workflow. To notice

[4] `http://www.boost.org/doc/libs/1_34_0/libs/statechart/doc/index.html`, Accessed: January 1, 2012.

[5] Here we abbreviate the IDs of the events from Figure 4 from "Ev_x_Desc" to "Ev_x".

misbehaviour in the workflow we first link emitted events from the services of the authentication process to the corresponding instance of the FSM and then check if said events correspond to the expected flow. This way we solve replay attacks, an attacked service invocation sequence, and missing events (see Section 3.2).

4.3 Profiling via Clustering

Creating a model that captures the behaviour of the infrastructure in a very granular fashion is difficult. This stems from the inherent complexity of packet-handling, e.g. flags for TCP packets are dependent on the implementation of the network stack of a system [24] and the amount of packets that are sent are dependent on the state of the network (e.g. congestion [25]). We use a heuristic approach to link infrastructure events to service events.

Linking Events to Create Profiles. The idea is a simple two-step procedure. (i) Relate all network events from a node to the last service call from that same node and (ii) relate all database CRUD events that involve the same identifier that was present in the service call to it. The time-window to relate the events to the service calls has to be set reasonably small otherwise the degree of relatedness diminishes. Infrastructure traffic linked in this way allows the creation of profiles of a service call type, e.g. a *network profile* and a *database profile.* The profiles themselves are obtained via clustering [26], which makes use of the inherent structure of data samples to group (cluster) common attributes and hence relate them. The network profile of a service call type, e.g. authentication, contains for example a clustering instance for each node in the network. A network profile, therefore, through its clustering instances, answers the questions *"Who communicates with whom normally?"* and *"How does traffic normally look like?"*.

In contrast, the database profile contains clustering instances for each user instead of nodes and pinpoints rare user behaviour in relation to a service call type. The profile types we chose reflect the dire need to monitor communication patterns among hosts (network profile) and CRUD activity (database profile) in health-care domains. Figure 5 summarizes how the layers are related. Workflow activities are linked to service call sequences. Then, infrastructure events are linked to service calls. Said events are then clustered to attain the profiles.

Fixed-Width Clustering. In fixed-width clustering [16], clusters are assigned a maximal width, ω, and instances lying outside, either create their own cluster or are assigned to another cluster. The idea is that small clusters reflect the η% of data instances which are anomalous.

Training and Detection. After an infrastructure event was linked to a service call, the training stage for clustering looks as follows,

1. We extract features from the infrastructure event, i.e. database and network features.

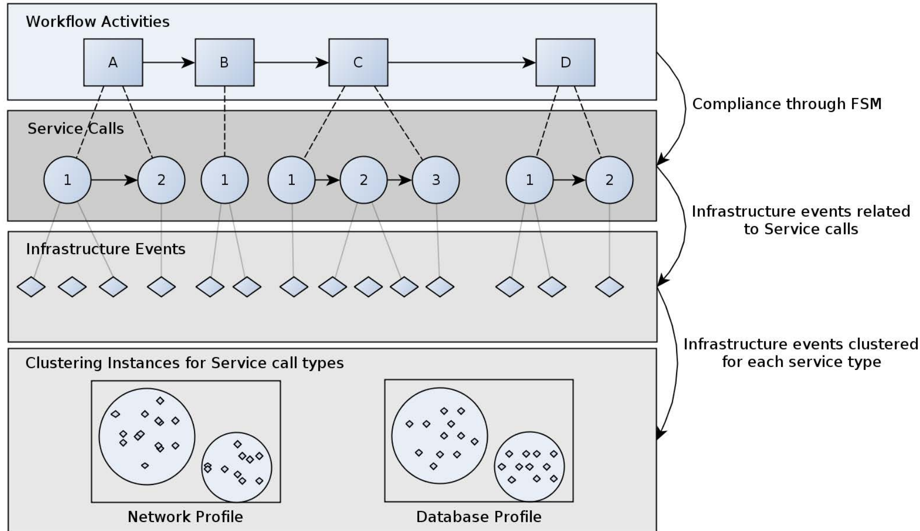


Fig. 5. Overview of connecting the layers

2. A feature vector that exceeds the width to any centroid of a cluster, creates a new cluster. Small clusters represent anomalous events after training.
3. During detection the feature vector of the infrastructure event in question is bound to the nearest cluster centroid. If such a cluster is anomalous, then so is the event.

Reduction of False-Positives. Depending on thresholds, and cluster-widths, false-positives occur. To reduce them, we introduce (a) white lists that allow clusters to be classified not to be harmful, e.g. communication to known non harmful websites, and (b) a threshold $\chi\%$ for the number of infrastructure anomalies that have to be detected in order for an alert to be raised. What can be learned from anomalies? Anomalies are always connected to hosts in the network. Therefore we mark a host as outlier if more than $\zeta\%$ of events linked to it are anomalous. Taking into account outlier nodes and their relation to services and workflows we obtain the possibility to trace a chain of implications to signal workflow issues. For instance, if a workflow relies upon services that in turn rely on nodes that are marked as outliers, the responsible for said workflow is alerted.

Examples. We explain how we solve the infrastructure attack example from Section 3.2, where a suspicious database event (Ev_5) is captured, via clustering. The three steps for the creation of a clustering instance for the service call type *Authorization Denied* operates as follows. In the first step, database events are captured that contain features, time, source, user, object, and CRUD type. Based on the linking heuristic they are associated to the corresponding service

call type, here *Authorization Denied*. Since the database profile consists of a clustering instance for each user, the *user* feature determines which clustering instance is chosen. In the second step, only one cluster was created, since the events were very similar. All infrastructure events for the service event *Authorization Denied* that were stored within the *database profile* (more specifically, within the clustering instance for UID_x) were read events (cRud) for UID_x with source PDP at a time from 8 in the morning to 7 in the evening. During detection (step three), a database event, Ev_5, at 2.01 a.m. is noticed. Due to the event being a database event it is added to the database profile. It is linked to the service call type *Authorization Denied* and due to the extracted user id, UID_x, it is added to the clustering instance for that user. Ev_5, an update PMR event (crUd) with identifier UID_x and object PMR_{12} at 2.01 a.m., is not common in the clustering instance. More formally, the features of the event, time, object, cause the whole data instance to exceed the maximal width of the existing cluster. Therefore, an alert from the *database profile* is triggered. The second infrastructure example from Section 3.2 is solved in a similar fashion, this time, the *network profile* is the source of the alert. Since the document consumer never initiated a communication with node D in the context of a retrieve service call during training, those events are outside any cluster and trigger an anomaly alert.

4.4 Architecture

In this section we identify the major components of our framework and give a brief description how the flow from event gathering to report takes place. As Figure 6 indicates, we distinguish among four component groups, *Nodes, Monitor, Core Components* and *View*.

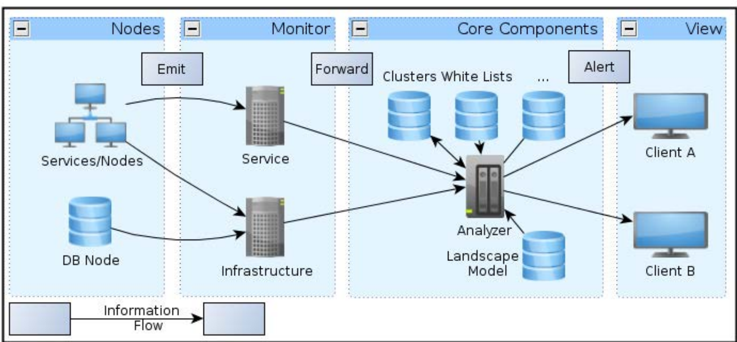


Fig. 6. Anomaly Detection Framework Architecture

1. The *Nodes* group represents the devices being monitored. We leverage the node monitoring capability of *Ossec*,[6] that allows to monitor, amongst other

[6] `http://www.ossec.net/`, Accessed: January 5, 2012.

things, database and network events on hosts through agents (sensors). At the same time to retrieve network data from routers we leverage the network information service provided by *Netflow* exporters.[7]

2. The *Monitor* group consists of web services that listen for incoming events. The Service Monitor receives all service events for which it is configured to accept. A service being part of a workflow, i.e. an identity provider, is configured to transmit service-call events (see Section 3) to the service monitor. The service monitor itself is built as a low-footprint web service and acts as an event sink for multiple sources. Node events, i.e. DB Activity, and network events, i.e. TCP/UDP, packets are sent to the Infrastructure Monitor. Network events are recorded as "Netflows" and forwarded to a Netflow-collector, our Infrastructure Monitor. The Infrastructure Monitor consists of a Web Service for node events and a Netflow-collector, *FLOWD*,[8] listening for Netflow traffic. The sensor then extracts the elements of the events we are interested in and relays this normalized data (if needed) to multiple analyzers. All the data is transmitted in the light-weight format called JSON,[9] to reduce working load for the monitors.
3. The *Core components* consist of the analyzer to which all normalized events are forwarded and a database for clusters, white lists, and the IT landscape model. The *IT landscape* model contains the information created during the modelling phase, in the machine-readable way XML Metadata Interchange (XMI)[27], but also concrete instances of running workflows in form of FSM instances. *White lists* are rules for traffic between hosts, which are considered not to be dangerous, e.g. connections going to and from "google mail". The standard white list encompasses the top 100 list of websites due to the Alexa.com rating site.[10] In the implementation the analyzer accesses white lists using MySQL. After the extraction of the features the *Clustering instances* are created. We leverage the C++ library *CLUTO*,[11] which provides a convenient API that allows to specify the desired clustering method and to additionally configure the selected method. Our first argument for the library was that it can handle large amounts of data efficiently.
4. The *View* unit consists of clients that are notified (push message) by the analyzer if an anomaly occurred. The clients connect to the analyzer over a web site that makes use of a HTML 5 (Web 2.0) interface allowing push messages via *WebSockets*.

5 Conclusion and Future Work

In the previous section we have presented a monitoring framework that incorporates modelling information to provide anomaly and compliance detection

[7] `http://tools.ietf.org/html/rfc3954`, Accessed: January 5, 2012.
[8] `http://www.mindrot.org/projects/flowd/`, Accessed: January 5, 2012.
[9] `http://www.json.org/`, Accessed: July 5, 2012.
[10] `http://www.alexa.com/topsites`, Accessed: January 5, 2012.
[11] `http://glaros.dtc.umn.edu/gkhome/views/cluto`, Accessed: January 5, 2012.

in complex IT landscapes. In contrast to other monitoring techniques this approach can dynamically link infrastructure events to various layers of abstractions, including services, workflow activities and also workflows. This allows a fine-grained detection of infrastructure event anomalies caused, or manifested, by service events. In addition, by leveraging workflow models, it is also possible to conduct compliance detection of technical workflows.

In our future work, we will focus on our implementation and evaluation efforts to create a fully functional anomaly detection framework that follows the scheme described in this paper. Therein, it will be essential to choose normalization parameters accordingly, i.e. for TCP packet sizes, to attain a proper clustering. We focus on improving on the linking heuristic, and also focus on extracting message payload from packets to further reduce false-positives. Another point for improvement is to improve on scalability, i.e. consider the "C10K" problem.[12] This can be achieved by building hierachies of monitors to filter out unnecessary events/traffic before it overloads our analysis system. We plan to address evaluation issues twofold, our clustering approach will be tested on the KDD benchmark whereas the whole framework will be tested in a real-world setting.

In the more distant future and from a network security point of view future work will be to classify outlying events and clusters to provide security experts with additional knowledge. A way is to detect attacks, e.g. ascertain that some cluster belongs to a brute-force attack, an XML attack, or similar. To address this, our anomaly detection framework can be augmented by other anomaly models, i.e. a payload model that detects string anomalities [28]. Based on the above research and the assumption that anomalous nodes and misbehaving workflows correlate we plan to incorporate predictive workflow compliance detection (similar to predictive SLA violations discussed in [29]) via statistical models (e.g. regression) and complex event processing (CEP) [30]. Its task is to predict workflow deterioration, e.g. stopped executions. For example, if multiple vital nodes show attack patterns and the amount of these patterns is rising over time, a deterioration can be predicted.

References

1. Portnoy, L., Eskin, E., Stolfo, S.: Intrusion detection with unlabeled data using clustering. In: Proceedings of ACM CSS Workshop on Data Mining Applied to Security, Philadelphia, PA (2001)
2. Leung, K., Leckie, C.: Unsupervised anomaly detection in network intrusion detection using clusters. In: Proceedings of the Twenty-Eighth Australasian Conference on Computer Science, vol. 38, pp. 333–342. Australian Computer Society, Inc. (2005)
3. Gu, G., Perdisci, R., Zhang, J., Lee, W.: Botminer: clustering analysis of network traffic for protocol-and structure-independent botnet detection. In: Proceedings of the 17th Conference on Security Symposium, pp. 139–154. USENIX Association (2008)

[12] `http://www.kegel.com/c10k.html`, Accessed: January 5, 2012.

4. Wang, W., Battiti, R.: Identifying intrusions in computer networks with principal component analysis. In: The First International Conference on Availability, Reliability and Security, ARES 2006, p. 8. IEEE (2006)
5. Garcia-Teodoro, P., Diaz-Verdejo, J., Macia-Fernandez, G., Vazquez, E.: Anomaly-based Network Intrusion Detection: Techniques, Systems and Challenges. Computers & Security 28(1-2), 18–28 (2009)
6. OMG: Omg uml specification, v2.0 (2005)
7. Breu, R., Innerhofer-Oberperfler, F., Yautsiukhin, A.: Quantitative assessment of enterprise security system. In: The Third International Conference on Availability, Reliability and Security, pp. 921–928. IEEE (2008)
8. Innerhofer-Oberperfler, F., Breu, R., Hafner, M.: Living security – collaborative security management in a changing world. In: Parallel and Distributed Computing and Networks/720: Software Engineering. ACTA Press (2011)
9. Breu, R.: Ten principles for living models-a manifesto of change-driven software engineering. In: 2010 International Conference on Complex, Intelligent and Software Intensive Systems, pp. 1–8. IEEE (2010)
10. Berre, A.: Service oriented architecture modeling language (soaml)-specification for the uml profile and metamodel for services (upms) (2008)
11. Popescu, V., Smith, V., Pandit, B.: Service modeling language, version 1.1. W3C recommendation, W3C (May 2009), http://www.w3.org/TR/2009/REC-sml-20090512/
12. van der Aalst, W.: Formalization and verification of event-driven process chains. Information and Software Technology 41(10), 639–650 (1999)
13. Mulo, E., Zdun, U., Dustdar, S.: Monitoring web service event trails for business compliance. In: 2009 IEEE International Conference on Service-Oriented Computing and Applications, SOCA, pp. 1–8. IEEE (2009)
14. Baresi, L., Guinea, S., Plebani, P.: WS-Policy for Service Monitoring. In: Bussler, C.J., Shan, M.-C. (eds.) TES 2005. LNCS, vol. 3811, pp. 72–83. Springer, Heidelberg (2006)
15. Erradi, A., Maheshwari, P., Tosic, V.: WS-Policy based monitoring of composite web services (2007)
16. Oldmeadow, J., Ravinutala, S., Leckie, C.: Adaptive Clustering for Network Intrusion Detection. In: Dai, H., Srikant, R., Zhang, C. (eds.) PAKDD 2004. LNCS (LNAI), vol. 3056, pp. 255–259. Springer, Heidelberg (2004)
17. Bertino, E., Ferrari, E., Atluri, V.: The specification and enforcement of authorization constraints in workflow management systems. ACM Transactions on Information and System Security (TISSEC) 2(1), 65–104 (1999)
18. Godik, S., Moses, T. (eds.): eXtensible Access Control Markup Language (XACML) Version 1.0 (February 2003)
19. Walker-Morgan, D.: Vsftpd backdoor discovered in source code. Website (2011), http://h-online.com/-1272310 (visited: July 4, 2011)
20. Hoglund, G., Butler, J.: Rootkits: subverting the Windows kernel. Addison-Wesley Professional (2006)
21. Peikari, C., Chuvakin, A.: Security Warrior. O'Reilly (2004)
22. Wells, J.: Computer fraud casebook: the bytes that bite. John Wiley & Sons Inc. (2008)
23. Kozen, D.: Automata and computability. Springer (1997)
24. McClure, S., Scambray, J., Kurtz, G.: Hacking exposed 6. McGraw-Hill (2009)
25. Allman, M., Paxson, V., Stevens, W.: RFC 2581 (rfc2581) - TCP Congestion Control. Technical Report 2581 (1999)

26. Tan, P., Steinbach, M., Kumar, V.: Cluster Analysis: basic concepts and algorithms. In: Introduction to Data Mining. Addison-Wensley (2006)
27. OMG: Omg xmi specification, v1.2 (2002)
28. Kruegel, C., Vigna, G.: Anomaly detection of web-based attacks. In: Proceedings of the 10th ACM Conference on Computer and Communications Security, pp. 251–261. ACM (2003)
29. Leitner, P., Wetzstein, B., Karastoyanova, D., Hummer, W., Dustdar, S., Leymann, F.: Preventing SLA Violations in Service Compositions Using Aspect-Based Fragment Substitution. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 365–380. Springer, Heidelberg (2010)
30. Nicolett, M., Litan, A., Proctor, P.E.: Pattern Discovery With Security Monitoring and Fraud Detection Techniques (2009)

Enhancing Model Driven Security through Pattern Refinement Techniques*

Basel Katt, Matthias Gander, Ruth Breu, and Michael Felderer

University of Innsbruck, Austria
{basel.katt,matthias.gander,ruth.breu,
michael.felderer}@uibk.ac.at

Abstract. Security requirements are typically defined at a business abstract level by non-technical security officers. However, in order to fulfill the security requirements, technical security controls or mechanisms have to be considered and deployed on the target system. Based on these security controls security patterns have to be selected. The MDS (Model Driven Security) approach uses security requirement models at a high level of abstraction to automatically generate security artefacts that configure security services. The main drawback of the current MDS solutions is that they consider just one security pattern for each security requirement. Current SOA and cloud services are scattered across multiple heterogeneous security domains. Partners and clients with different security infrastructures are changing continuously, which requires the support of multiple patterns for the same security service. The challenge is to provide configurable security services that can support different patterns. In order to overcome this shortcoming we propose a framework that integrates pattern refinement to the MDS approach. In this approach a security pattern refinement layer is added to the traditional MDS layers. The pattern refinement layer supports the configuration of one security service with different patterns, which are stored in a pattern catalog. For example, our approach enables the generation of security artefacts that configure a non-repudiation service to support both *fair non-repudiation* and *naive non-repudiation* patterns.

1 Introduction

MDS (Model Driven Security) aims at closing the gaps (i) between abstract business oriented security requirements and low level security artefacts, and (ii) between functional models and security models. The latter is done by introducing security requirements on the functional models. The former can be achieved using transformation functions that generate security artefacts from abstract security requirements. Another security engineering technique that is used in solving security problems involves *security patterns*. A security pattern is a well-understood solution to a common security problem or threat. In other words, a security pattern embeds security expertise in the form of tested and worked solutions.

* This work is supported by QE LaB - Living Models for Open Systems (FFG 822740), COSEMA - funded by the Tiroler Zukunftsstiftung, SecureChange (ICT-FET-231101) EU project, and SECTISSIMO (P-20388) FWF project.

B. Beckert et al. (Eds.): FMCO 2011, LNCS 7542, pp. 169–183, 2013.
© Springer-Verlag Berlin Heidelberg 2013

One of the main goals of the security engineering paradigm is to provide security solutions that fulfill the needs stated in abstract security requirements. In order to cope with such security requirements, security controls have to be applied. We define security controls as security procedures or mechanisms that are designed to protect against threats, reduce and limit vulnerabilities, detect malicious incidents, and facilitate recovery. The realization of a security control is done through the development of security services. Such security services should offer a general purpose security functionality that needs to be configured based on the platform and domain information and a well-known security pattern. For example, an authentication requirement can be met using one of the standard authentication mechanisms, e.g. four-eyes principle, single sign-on, or brokered-authentication. Furthermore, a specific pattern has to be chosen for the implementation of that control, for example, a *direct authentication* pattern is suitable for the case when the authenticated users belong to the same security domain. Finally, the deployment of security services and mechansims depends on the system architecture, for which the security solution has been designed.

State of the Art. According to the paradigm of MDS, abstract security policies are used to extend functional models with security concerns. The abstract models (or artefacts) are then transformed into platform-specific artefacts by code-generation methods. Those artefacts configure the target platform. The approaches presented in [1,7,14,10] deal with similar processes to realize security modeling concepts. The focus of current security modeling approaches is to model abstract security policies and transform them into executables. In most of the cases, the executables are XML-based security policies, such as authentication, authorization and non-repudiation policies. For example, Basin et. al. [1], have generated authorization policies to configure access control infrastructures for J2EE applications. Fumiko et. al. [14], have added security annotations in the system models and transformed them into authentication policy assertions for WS-SecurityPolicy. Hafner et. al. [7], have defined security requirements in business process models and generated XACML code from abstract requirements. Ulrich Lang et. al. [10] have applied model-driven security to CORBA platform, and generated security policies from models based on a Policy Definition Language (PDL).

Limitations of Current MDS. A common assumption in these approaches is that the security services, which use the policies generated from models, are already deployed at the target platform. Those security services realize certain pre-defined security patterns and mechanisms. For example, an authentication security service implements the *Brokered Authentication* pattern or an authorization service implements the *RBAC (Role-based Access Control)* pattern. As a result, an authentication service, which realizes the brokered authentication pattern cannot implement a direct authentication or identity federation pattern. Similarly, an authorization service, which realizes the RBAC pattern, cannot implement the attribute-based or context-based access control patterns.

The problem is that in real SOA systems, different partners and service providers from different security domains and heterogeneous security infrastructures communicate with each other. Changing a partner, launching a business service with a new partner or acquiring a new costumer, all with different security needs, constitute the

dynamic nature of a SOA business model. This requires that security services have to realize various patterns depending on the security domain of requester. For example, a service provider will implement a direct authentication pattern if a service requester's identity is validated from the local identities. The provider will implement a brokered authentication pattern, if it relies on the authentication decision of an external Identity Provider. Alternatively, the provider may rely on Identity Federation and Single-Sign-on patterns, if a service requester's identity is federated among multiple domains. This shows that a security service has to be configurable to realize a variety of patterns. Using model-transformation techniques, the security service configurations can be generated from the models. However, the current security modeling approaches rely on fixed/hard-coded security services using pre-defined patterns. As a result, the security services at the target platform can not be configured to use other patterns. This makes these approaches difficult to extend.

1.1 Contributions

In this paper we propose to overcome the shortcomings of the current MDS approaches by integrating a pattern refinement process in an MDS framework. This is done by introducing a pattern refinement layer that resides between the PIM (Platform Independent Model) and PSM (Platform Specific Model) layers. Our framework enables the semi-automatic generation of security artefacts for different patterns from the same security requirement. We discussed a general MDS approach without the consideration of pattern refinement in our previous work [7], however, pattern refinement integration is innovative in this work.

Applying this approach requires an architectural solution that provides the design principles of a general purpose and configurable security services. In previous work [8], we proposed the SeAAS (Security As A Service) architectural solution that aims at meeting the end point security problems by decoupling security services from the business end points. Security services according to the SeAAS methodology are offered centrally in each security domain and can be configured to support multiple patterns. The discussion of SeAAS is out of scope of this work. In principle, our new MDS approach is general and can be applied for different architectural designs, however, SeAAS is taken as a use case.

2 Case Study

The case study is taken from the SecureChange EU-FP7 project. The use case deals with a web service for retrieving newsfeeds as introduced in section 7 of Deliverable 2.3[1]. As depicted in Figure 1 the HOMES case study consists of several infrastructural components, which are deployed in the domains of three actors:

1. The Customer or Service Requester (SR) uses the Home Gateway device to connect to the internet. He may purchase, install, and use additional services at home.

[1] `http://securechange.eu/content/deliverables`

2. The Network Operator (NO) owns and operates the infrastructure necessary to provide connectivity for customers. He advertises, installs and manages additional value-added services. The latter may be provided by Third Party Service Providers.
3. The Home Gateway device (HOMES) is placed at the customer's home within his security domain. It is owned by the network operator and usually rented to the Customer. The HOMES Gateway device is the service platform for the case study.
4. Third Party Service Providers (SPs) offer additional services for customers. They are independent from the NO and have a commercial agreement with the NO.

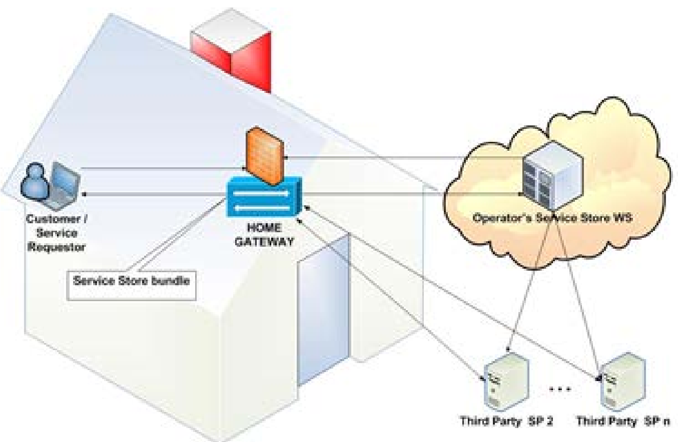


Fig. 1. Message flow within the HOLMES case study

In the case study the SR continuously consumes newsfeeds from distinct trusted third party SPs. This communication is controlled via the HOME Gateway, which then handles the communication to the Service Provider with respect to the corresponding security requirements. In more detail, first the customer sends a request to the HOME Gateway which then contacts the NO in order to retrieve a corresponding policy file that contains information about the security service and the security policy required. After performing the security tasks specified in this policy, the HOME Gateway subsequently requests the feeds from the Service Provider. The response is sent back in the same way.

Besides the message workflow also the security requirements within the process of requesting newsfeeds are illustrated in Figure 2. The figure indicates that both messages between the HOME Gateway and the Service Provider cannot be repudiated (non-repudiation requirement). The business partners (HOME Gateway and the SPs) are located in different domains. Within each of these domains there is a SeAAS engine deployed which is responsible to enforce the security requirements. Consider, for instance, the HOME Gateway that communicates with multiple companies from which it uses services ($\mathrm{SP}_1, \ldots, \mathrm{SP}_n$) (cf. Figure 2). Therefore the HOME Gateway has to fulfill a multitude of different security requirements which result in different business policies (at most n). Even if the HOME Gateway has to fulfill the same security requirements

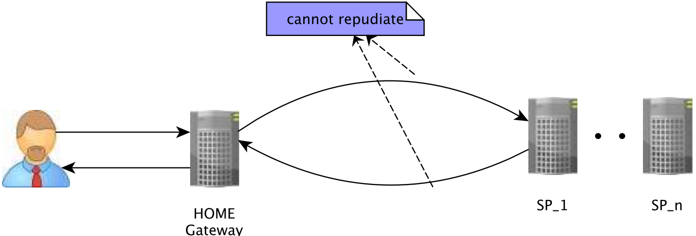


Fig. 2. Policy dependencies between the actors

for all of the SPs their technical policies or the used security pattern might still be different. It can be the case that one SP only supports a naive non-repudiation while another one requires fair non-repudiation. It can be noticed that the MDS and the architectural solution should enable the configuration of a security service with different patterns. As aforementioned, we have proposed an architectural solution (SeAAS) and in this paper we propose the MDS solution to create the required configurations.

3 Methodology: SECTET Framework

As shown in Figure 3 (left), the traditional MDS approach consists of two design time layers and one run time layer. In PIM models, security requirements are attached into the functional models. Those models are abstract and independent from the run time platform. PSMs, on the other hand, represent the platform specific functional and security models. For example, if a platform runs a BPEL engine, the workflow model in this layer is a BPEL model. Examples of policy models are XACML [11], or PGP[2] models. M2M (Model to Model) transformations are used to generate PSM models out of the PIM models. Finally, ISM (Implementation Specific Models) represent the run time system and contain the actual implementation code and configuration files deployed on the system.

The SECTET framework enhances the traditional MDS approach with a pattern refinement process. This is done by filling the gap between the PIM layer and the PSM layer with a pattern refinement layer and the introduction of a pattern catalog that stores models about the supported patterns. The result is a three layered framework that contains, besides the PIM and the PSM layers, a new layer called PRM (Pattern Refinement Model). In this layer different security patterns, which meet the same requirement, can be chosen from a pattern catalog and refined to generate configurations for security services. This requires two security artefacts to be generated. First a security policy and second a workflow that indicates the behavior of the pattern. we call the policy a *pattern policy* and the workflow a *pattern flow*.

3.1 Platform Independent Models: SECTET PIM Metamodel

At the first PIM layer the SECTET framework defines a metamodel, that enables modeling security enhanced workflow (or business process) models. This language consists

[2] http://www.pgpi.org/

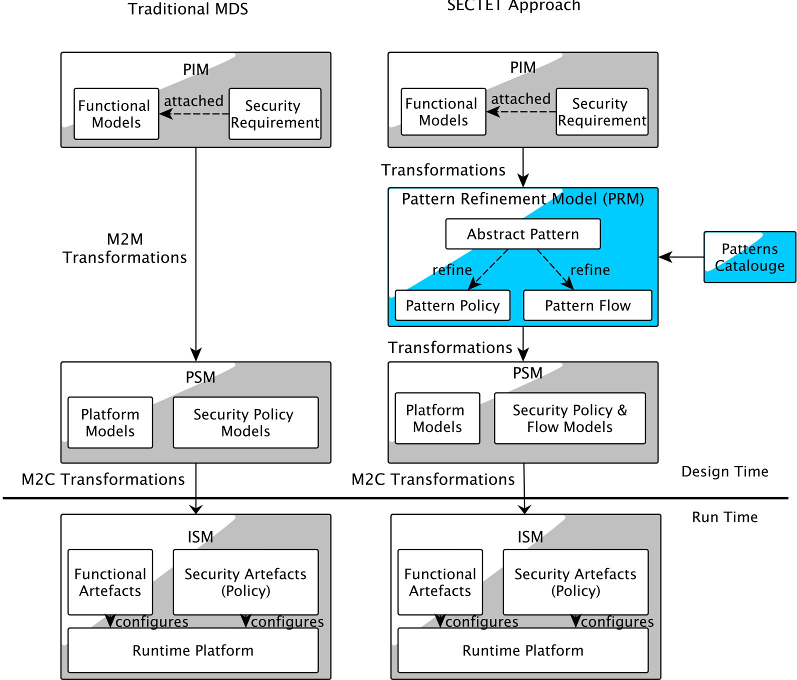


Fig. 3. SECTET Approach

of the following views (cf. Figure 4). This metamodel is similar to the SECTET DSL presented in [6].

- Workflow view: Workflows or business processes are the functional models that are considered in the SECTET framework. The main components of the workflow view is a process that consists of several activity nodes, object nodes and partners. Activity nodes represent the operations that each partner offers and object nodes represent the messages that are exchanged.
- Interface View: The interface model contains information about the services that are deployed by different partners and the documents that are sent or received. The services consist of one or more operations. Furthermore, each operation is associated with a request and a response, both of *Message* type.
- Requirement View: The security requirement view consists of both *ServiceReq* as well as *MessageReq* classes. The first indicates the security requirements that can be attached to a service and the latter indicates the security requirements that can be attached to a message. For example, Figure 4 indicates that the main security requirement that can be defined for a service is *Authentication*, while the security requirements that can be defined for a message are *Monitor, Integrity, Confidentialiy*, and *Non-Repudiation*.

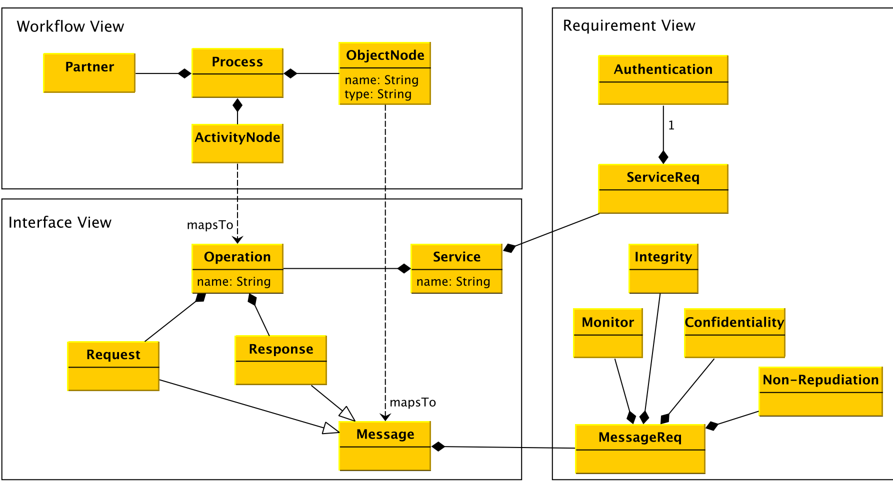


Fig. 4. SECTET PIM metamodel

3.2 Pattern Refinement Models: Pattern Models

The SECTET framework aims at closing the gap betweent PIMs and PSMs using pattern refinement techniques. Security patterns solve specific security problems in a specific context. For example, an *authorization pattern* ensures that only people with permissions are allowed to access protected resources. The *Authentication pattern*, on the other hand, tackles the problem of identity and message validation. Similar to models in traditional MDS approaches, patterns can be defined as abstract patterns and concrete or technical patterns (cf. Figure 5). The first step in our pattern refinement process is to map each security requirement to an abstract security pattern. Second, based on the target architecture of the runtime system an architectural security pattern is selected. For example, either a *direct authentication* or a *brokered authentication* can be selected from an abstract pattern of authentication. From an abstract non-repudiation pattern a *naive non-repudiation* or a *fair non-repudiation* pattern can be selected. After a specific architectural pattern is selected two refinements are applied on the selected pattern. The first refinement aims at generating a workflow that indicates the behavior of the security service when the selected pattern is applied. The second aims at generating a security policy associated with this pattern.

Pattern Catalog. Patterns that are used in our pattern refinement process are stored in a pattern catalog. This enables the reuse of established solutions for well-known problems. Furthermore, the extension of the system to accomodate new patterns is done by adding a new pattern to the catalog. For each architectural security pattern, models that represent the flow of actions, i.e., the protocol of the pattern, as well as the security policy that is required are stored. When the architectural pattern is chosen, information related to both pattern flow and pattern policy is provided.

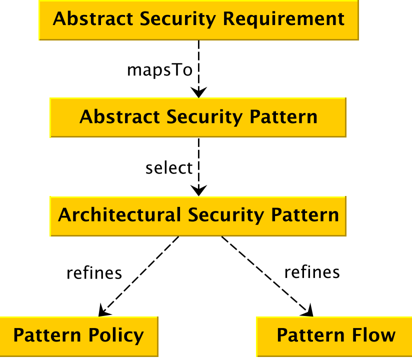


Fig. 5. Pattern refinement models

3.3 Platform Specific Models

The third layer is similar to the third layer in the traditional MDS approach, which contains the platform specific models. Models in this layer describe the system on its target platform. For example, workflow models in the second layer can be represented as BPEL4WS[3] in case the platform contains a BPEL based workflow engine. It can be noticed that, in this layer we consider both the security policy and the workflow that represent the flow of actions executed by each pattern. In this way multiple architectural patterns can be supported by one security service. To summarize, the service can be configured with the workflow of the architectural pattern as well as with the security policy.

3.4 Implementation Specific Models

This layer represents the runtime system and contains the reference architecture that acts as the runtime environment. All generated security and functional artefacts and code parts are part of this layer. The transformations that result in artefacts in this layer are mainly M2C (Model to Code) transformations.

4 Case Study Example

In this section we show how the SECTET's security engineering framework is used to solve the problem discussed in the case study in Section 2. Using our framework, it is possible to generate, for each security requirement, security and functional artefacts for different security patterns.

[3] `http://www.ibm.com/developerworks/library/specification/ws-bpel/`

4.1 Modeling Functional Workflow with Security Requirements

The SECTET approach requires at the first layer to model the functional workflow that represent the exchange of messages between different partners and the associated security requiremets. The workflow model is represented as an UML activity diagram and the services as well as the messages exchanged (forming together the interface view) are represented as UML interfaces and UML class diagrams, respectively.

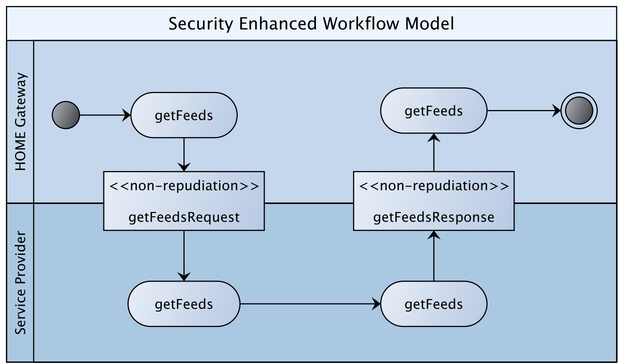


Fig. 6. Workflow model enhanced with security requirements

Figure 6 illustrates an activity diagram that considers two partners in the whole workflow, namely, the HOME gateway and the service provider. The example illustrates the task of retrieving feeds. A customer behind a HOME Gateway device wants to retrieve newsfeeds from the serviced provider. Therefore, the HOME Gateway sends the corresponding request to the service provider. The main service that is shown in this diagram is *getFeeds*. Two messages are exchanged, which are *getFeedsRequest* and *getFeedsResponse*. The security requirements are attached with the data object that flows between the particular activities. The figure also shows that for both requesting message and responding message *non-repudiation* is specified as a security requirement.

As discussed in Section 3, the second view of the metamodel in the first PIM layer is the interface view. The interface view consists of the operations that are provided as service and the messages that are exchanged. Figure 7 illustrates the operations, or services that the service provider offers, which is one service called *getFeeds*. It can be seen that there is one security requirement attached to this service, namely, authentication. Figure 8, on the other hand, represents the messages that are exchanged.

Fig. 7. Services and their attached security requirements

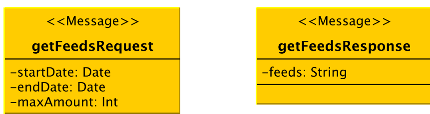


Fig. 8. Messages and their attached security requirements

The security requirements attached to these messages are shown in the workflow model. To summarize, the PIM models contain information about the workflows, including the services and the messages exchanged (functional models), and the security requirements attached to those models.

4.2 Modeling Pattern Refinement

The second layer in our framework is the pattern refinement layer. Figure 9 shows the abstract security patterns that are supported by our framework. These patterns correspond to the abstract security requirements that can be defined in the PIM model. The first step is to map each security requirement to an abstract security pattern. For example, the non-repudiation requirement defined for the *getFeedsRequest* and *getFeedsResponse* messages is mapped to the abstract non-repudiation pattern. The second step is to select one of the architectural patterns supported for this specific abstract pattern. This decision is made based on the target architecture and the pattern supported by the security service at the SP site. Figure 10 shows two architectural patterns supported for the abstract authentication pattern, namely, *naive non-repudiation* and *fair non-repudiation* pattern.

The third and last step is the refinement step, which includes two refinements, the policy and flow refinement. Figure 11 shows the fair non-repudiation pattern flow that is defined in the pattern catalog. The flow model indicates how the non-repudiation service deployed in service requester communicates with the SP and the trusted third

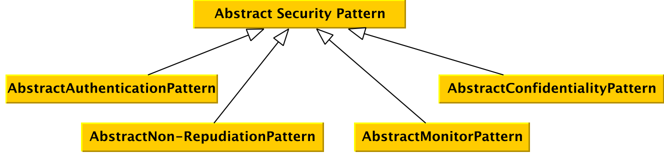


Fig. 9. Abstract security patterns

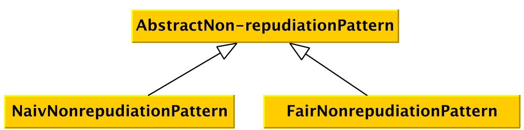


Fig. 10. Non-repudiation architectural patterns

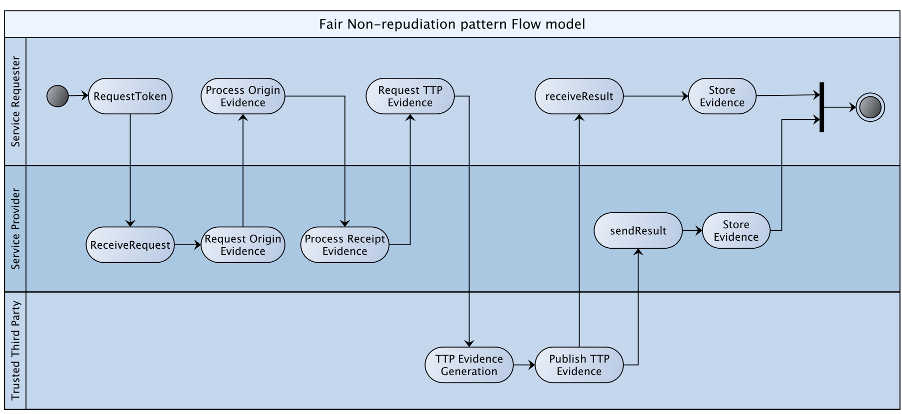


Fig. 11. Fair Non-repudiation pattern flow

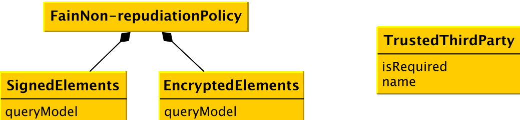


Fig. 12. Fair Non-repudiation pattern policy metamodel

party in order to ensure fair non-repudiation. This flow is an abstract workflow model which requires some concrete information based on the use case, which will be provided when the pattern flow is refined. We assume that the only information that must be provided in the flow refinement step is the trusted third party. This means that after the fair non-repudiation pattern is selected, the refinement of its pattern flow requires to add information about the trusted third party. Similarly, we assume a simple policy for fair non-repudiation that indicates which elements must be signed and which elements must be encrypted in the message for the evidence. Figure 12 shows the non-repudiation policy model which indicates that both the encrypted and the signed elements for the query model must be specified. The query model represents the query message and indicates the getFeedsRequest class in our example. The refinement of the pattern policy adds information about the elements in the request message that must be encrypted and signed. These elements are selected from *startDate*, *endDate*, and *maxAmount*.

4.3 Platform Specific Model and Generated Artefacts

After performing the second refinement step we get refined models for a fair non-repudiation policy (These are pattern policy model and pattern flow model for the fair non-repudiation pattern). The last layer in our design related models is the platform specific models. Our runtime system is based on the SeAAS architecture (for details we refer to [8]). The security services in the SeAAS architecture are developed based on a BPEL engine. Furthermore, WS-* standards are used for encoding security as well as

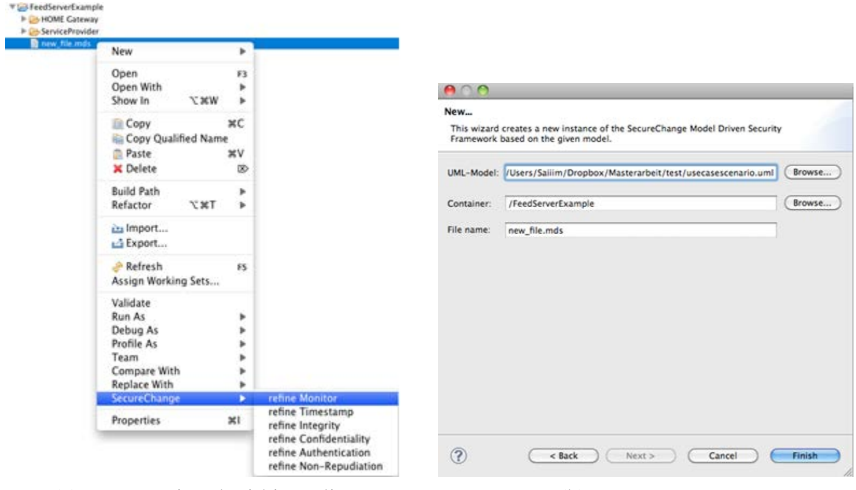
functional policies. Therefore, the platform specific workflow models are BPEL models and the platform specific fair non-repudiation service is based on a WS-Policy model. More information about our architectural design based on SeAAS can be found in [8].

5 Prototypical Implementation

We have developed a prototype that realizes the framework components that we discussed before. In this section we present our prototype and show the steps that are performed in our framework and give examples of screenshoots for the graphical user interface. The prototype is implemented as an Eclipse plug-in. The GUI consists of two main parts. An import interface for importing the UML models and a transformation interface for performing the transformation and refinement steps.

5.1 UML Models Import

The model import interface is responsible to import different UML models. The main SECTET (meta-)models for the different layers, i.e. the PIM metamodel shown in Figure 4, the refinement metamodels and the pattern models stored in the pattern catalog (e.g., in Figures 9, 10, 11 and 12) are previously stored in our Eclipse plug-in. These models can be changed or extended, e.g., if a new pattern is plugged into the pattern catalog or a new requirement is introduced. The case study models can be created by any UML-model editor and imported to our prototype through the import GUI shown in Figure 13. These models include the workflow activity diagram (Workflow view) and class diagrams of the interface view, i.e., models shown in Figures 6, 7 and 8.



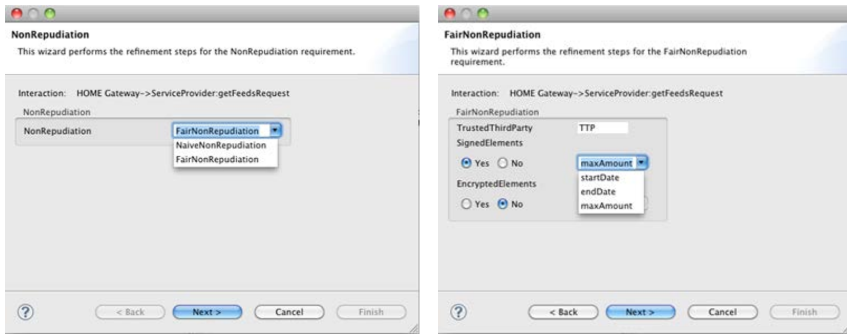
(a) Import wizard within Eclipse.

(b) Import screen.

Fig. 13. Import interface for importing UML models

5.2 Selection and Refinement Wizards

The second main interface in our implementation is the selection and refinement wizards. The selection and refinement wizards are automatically generated from the pattern models. For example, from the non-repudiation metamodel shown in Figure 9 the selection wizard shown in Figure14(a) is generated, which enables the selection between the different non-repudiation pattern supported in our pattern catalog. The first and second refinement steps are integrated in one wizard shown in Figure 14(b). Based on the flow and policy patterns of non-repudiation this wizard is automatically generated and enables entering information required for generating the pattern workflow as well as the pattern policy of our case study.



(a) Selection wizard for abstract non-repudiation patterns.

(b) Refinement wizard for fair non-repudiation pattern.

Fig. 14. Selection and refinement wizards

6 Related Work

Different approaches have been proposed in the literature that deal with modeling of security requirements using security patterns, protocols and code generation methods. In [19,18], Wolter et. al. have modeled security goals for cross-organizational business processes. They have considered SOA-based federated environment, which comprises multiple independent trust domains. Other work within the Security Engineering community deals with specification of security requirements in the context of formal methods. Examples are, the UML extension UMLsec [9] together with the AUTOFOCUS tool [17] and the PCL approach [2]. Few groups deal with aspects of code generation in the context of secure solutions. Among these are the groups of Basin et. al. [16,1] and Ulrich Lang [10]. Both approaches present frameworks for high-level specification of access rights including code generation; the former in J2EE and the latter in CORBA environments. Satoh et.al. [14], have solved authentication scenarios in an environment using identities, which are federated among multiple domains and generated WS-SecurityPolicy assertions for IBM-WAS.

Security patterns have been extensively used for modeling security requirements [15,12]. In [3,13] D. G. Rosado et. al. have discussed the high- and mid-level abstractions of security patterns and architectural security patterns. N. A. Delessy et. al. [4], in their research have proposed pattern-driven security process for SOA applications. F. B. Medina et. al. [5] have introduced abstraction of security patterns, based on the problem space and architecture of patterns.

The main problem of these modeling aproaches is that they assume one security pattern for the generated security artefacts and fixed and hard-coded security services. The proposed security modeling framework SECTET is an extension of the original SECTET framework presented in [6], which makes it possible to support different patterns for the same security requirement based on a pattern catalog that stores pattern models. Original SECTET have addressed security aspects in inter-organizational workflows and transformed security extensions into policies to configure the target architecture. In pattern enhanced SECTET, we introduce a further layer of abstraction between secure business models and platform-specific artefacts. The security requirements are mapped to abstract security patterns, which are used to select a suitable architectural security pattern. Finally, two refinements are executed to generate pattern flow and pattern policy models, which are used to generate configurations for security services of a SeAAS-based target architecture.

7 Conclusion

In this paper, we have proposed the SECTET framework, for security modeling using a security pattern refinement process. In pattern enhanced SECTET, we have extended our previous SECTET framework in particular and current security modeling approaches in general, by two dimensions. First, we introduced a Pattern Refinement process between abstract security requirements and code. With this we are able to transform the high level security patterns to concrete security patterns for a target security architecture using configurations generated from models. Second, our target platform i.e., SeAAS (Security As A Service) architecture is based on the principles of SOA applications, which enforces security with decoupled, dedicated and shared security services in a security domain. We found that the current security modeling approaches rely on hard-coded security services. With the proposed extension, based on pattern refinements, the security services executing at the target platform can be configured to use a variety of security patterns and mechanisms. We have illustrated our approach for cross-domain non-repudiation requirement of the HOME case study of SecureChange EU-FP7 project. We have refined the non-repudiation pattern for SeAAS architecture as a target platform and generated WS-SecurityPolicy to configure the security services. In future, we intend to investigate more complex security scenarios and solve them using our approach. Furthermore, formal methods can investigated for the purpose of the verification of the soundness of transformation process, as well as the analysis of the generated run-time security policies.

References

1. Basin, D., Doser, J., Lodderstedt, T.: Model Driven Security: From UML Models to Access Control Infrastructures. ACM Trans. Softw. Eng. Methodol. 15(1), 39–91 (2006)
2. Datta, A., Derek, A., Mitchell, J., Pavlovic, D.: A derivation system and compositional logic for security protocols. J. Comput. Secur. 13(3), 423–482 (2005)
3. David, R., Carlos, G., Fernandez-Medina, E., Piattini, M.: Security patterns and requirements for internet-based applications. Internet Research 16(5), 519–536 (2006)
4. Delessy, N., Fernandez, E.B.: A Pattern-Driven Security Process for SOA Applications. In: ARES 2008: Proceedings of the 2008 Third International Conference on Availability, Reliability and Security, pp. 416–421. IEEE Computer Society, Washington, DC (2008)
5. Fernandez, E.B., Washizaki, H., Yoshioka, N.: Abstract Security Patterns. In: SPAQu 2008 - 2nd Int. Workshop on Software Patterns and Quality (2008), `http://patterns-wg.fuka.info.waseda.ac.jp/SPAQU/`
6. Hafner, M.: SECTET A Domain Architecture for Model Driven Security. PhD Thesis (November 2006)
7. Hafner, M., Breu, R.: Security Engineering for Service-oriented Architectures. Springer (October 2008)
8. Hafner, M., Memon, M., Breu, R.: SeAAS - A Reference Architecture for Security Services in SOA. Journal of Universal Computer Science 15(15), 2916–2936 (2009), `http://www.jucs.org/jucs_15_15/seaas_a_reference_architecture`
9. Juerjens, J.: Secure Systems Development with UML. Springer (2004)
10. Lang, U., Schreiner, R.: Developing Secure Distributed Systems with CORBA. Artech House, Inc., Norwood (2002)
11. OASIS. Extensible Access Control Markup Language (XACML) (2006), `http://www.oasis-open.org`
12. Rodriguez, A., Fernandez-Medina, E., Piattini, M.: A BPMN Extension for the Modeling of Security Requirements in Business Processes. IEICE - Transactions on Information and Systems E90-D(4), 745–752 (2007)
13. Rosado, D.G., Fernandez-Medina, E., Piattini, M.: Comparison of Security Patterns. IJCSNS -International Journal of Computer Science and Network Security 6(2B), 139–146 (2006)
14. Satoh, F., Nakamura, Y., Ono, K.: Adding Authentication to Model Driven Security. In: ICWS 2006: Proceedings of the IEEE International Conference on Web Services, pp. 585–594. IEEE Computer Society, Washington, DC (2006)
15. Schumacher, M.: Security Engineering with Patterns: Origins, Theoretical Models, and New Applications. Springer-Verlag New York, Inc., Secaucus (2003)
16. Lodderstedt, T., Basin, D., Doser, J.: SecureUML: A UML-Based Modeling Language for Model-Driven Security. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 426–441. Springer, Heidelberg (2002)
17. Wimmel, G., Wisspeintner, A.: Extended Description Techniques for Security Engineering. In: Dupuy, M., Paradinas, P. (eds.) Trusted Information. IFIP, vol. 65, pp. 469–485. Springer, Boston (2002)
18. Wolter, C., Menzel, M., Christoph, M., et al.: Model-driven business process security requirement specification. J. Syst. Archit. 55(4), 211–223 (2009)
19. Wolter, C., Menzel, M., Meinel, C.: Modelling Security Goals in Business Processes. In: Modellierung, pp. 197–212 (2008)

Project Zeppelin: A Modern Web Application Development Framework

Leigh Griffin, Peter Elger, and Eamonn de Leastar

Telecommunications Software and Systems Group,
Waterford Institute of Technology, Waterford, Ireland
{lgriffin,pelger,edeleastar}@tssg.org

Abstract. Application Platforms, by which we mean the programming languages, libraries, frameworks and associated run time support, are central to the modern development experience. They are often imbued with an ethos, value set and engineering approach that carries through the full lifecycle of the platform itself, steering its development and evolution through the various challenges - both technical and commercial - it must surmount in order to survive. Anecdotal evidence would suggest that these platforms have a lifespan of approximately 10 years - after which they enter a gradual decline. The reasons for this decline vary, including commercial shifts, new (or rediscovered) thinking and changes in the underlying technology. The authors believe that two of the major platforms in use today - J2EE and .NET - may be about to enter this declining phase. The major factors contributing to this decline; including considerable complexity, significant disjunction in the developer experience and major challenges in meeting the demands of the modern, predominantly mobile, social web. A new application platform, dubbed Zeppelin, architected to programatically meet the challenges of the Future Internet is presented.

Keywords: Eternal System, Javascript, Node.js, Web Application.

1 Introduction

New applications and services are increasingly being developed so that they can be deployed into the cloud, be that Amazon Web Services (AWS), Microsoft Azure, special purpose clouds such as those emerging for level 3 secure health care data, home grown clouds or clouds provided by other third-party vendors. As services are deployed into third-party clouds the cost of operating these services change in their profile to become more opex (operational expenditure) based, as they typically incur monthly, recurring fees based on their usage of cloud resources. These opex costs drive new requirements for services such as efficiency and quality, in terms of using cloud resources, as an inefficient service can generate substantial additional, and unneeded, costs. A poor quality service, as determined by resource consumption, may inadvertently spike costs by

B. Beckert et al. (Eds.): FMCO 2011, LNCS 7542, pp. 184–198, 2013.
© Springer-Verlag Berlin Heidelberg 2013

an order of magnitude whilst handling a relatively small number of clients. The challenge for todays services can be summarised by what has become known as the C10K problem - how can a service deal with 10,000 concurrent clients on a single system and how can this scale horizontally to handle 20K or 30K simultaneous connections? This is the classic traffic spike problem, whereby a service or website suddenly has to handle 10,000 web browsers, or 10,000 mobile phones all asking for the same key piece of data. The phenomenal rise of social networking and the associated applications provisioned across it has made this usage scenario a daily reality for service management. As a result, performance of concurrency, or threading, becomes a key issue for modern services in the cloud and in social networks. Failure to deal with the C10K problem often results in service failure in production. Related to this are the challenges of scalability and robustness i.e. how can a cloud service gracefully scale as the service takes off, without failure as its traffic grows and also how can we ensure that instead of proportionally increasing opex costs, the service exhibits economies of scale.

These are not trivial challenges and current service stacks such as the LAMP (Linux, Apache, MySQL, and PHP) stack, service creation frameworks such as Enterprise Java or Ruby on Rails, and their resultant service management technologies are destabilizing in the face of these demands [4]. Within this paper we will explore the challenging world of provisioning services within the cloud, examine the technological shift away from traditional approaches and outline an approach for developing a modern web application framework. This paper is broken down into six sections. This section, the introduction, serves as the first. The technical context examining the change in the environment is presented in section two. The third section examines the challenges of concurrency in modern programming. Section four discusses development frameworks and platforms to host services and the strategies involved. The fifth section looks at Project Zeppelin, outlining the design decisions and architecture. The sixth, and final section, discusses our future work and conclusions.

2 Technical Context

2.1 Web Evolution

The evolution of the Web has been well charted in recent years. With the coining of the term Web 2.0 (2004) [8] a useful starting point, there is a reasonably comprehensive understanding of how the web as a platform has progressed since. This has included the stabilisiation of web services protocols, the rise of User Generated Content, the proliferation of Mobile web, the advent of the Smart Phone/Device/Tablet and the App Store Model. However, the underlying tools, architectures and development practices have also been through an overhaul over this period and have made this innovation cycle possible. In particular, there has been a marked shift from the traditional enterprise stack (EJB2, .NET), and high ceremony development methods (RUP) to a more agile approach (XP), with a full embrace of more open tools and frameworks. Sometimes termed lightweight approach, this shift has included rapid evolution in web frameworks

(Spring, OSGi) and the arrival of highly productive variations on these, most notably Ruby on Rails and its derivatives, usually bound to a relational database (MySQL). Coupled with this evolving stack, web browser performance, stability and capability has improved dramatically. The client side web is now clearly formed by the nexus of HTML, CSS and JavaScript, with the latter in particular the lynchpin of significant innovation in the usability, power and flexibility of web applications. A set of JavaScript libraries (jQuery and others) has radically altered the usage patterns within the browser, unleashing unsuspected features in a robust environment. All three are sometimes grouped under the term HTML5, which in addition includes standardised approached to geo-location, 2D and 3D graphics, offline services, standard communications channels and more. Although not quite accurate, the term HTML5 usefully encapsulates the reach and ambition of the latest wave of browsers, and would seem to have significant momentum from all major players, hardware and software and infrastructure.

2.2 Revolution

There are signs, however, that this evolutionary approach may have run its course. The lightweight development stack of Relational Database, Component Service/Framework + Template Engine, all running on a Linux back end (a variant of the so called LAMP stack), is encountering a major shift in the underlying infrastructure - the arrival of the cloud. In particular, cloud based services coupled with advances in virtualization, have altered the principles around which applications have been architected to date. When this is also combined with various models for smart phone/tablet development, there is an argument that we are entering into another inflection point, comparable to the one foreseen in 2004. What this particular movement will lead to is as yet unclear. However, it seems certain to yield new opportunities in services, mobility, flexibility and productivity in application development and deployment. In this context, there are signs of disruption within the current development stack. Although significant stability has been achieved since 2004, many of its tenets are now being called into question:

Database. The dominance of the relational database is no longer a given. The NoSql movement [15] is gathering pace with many open implementations of this broader, and perhaps more scalable architecture for the data store (MongoDB, CouchDB). When coupled with Googles Map/Reduce, it may be possible for more highly capable and intelligent systems to be constructed at a fraction of the cost of traditional relational systems.

Middleware. Having already preceded though a series of major shifts over the past decade (rise and fall of Object Request Brokers, rise and fall of EJB, rise and stabilization of web frameworks), middleware is a useful touchstone when assessing the state of software and services. Evidence is mounting that the sheer complexity of current enterprise stack (J2EE, .NET) is causing profound limitations in the scale and reach of applications thus architected. The lightweight

stack, evolved in some sense as an alternative to the traditional stack, may have reached its peak in Heroku, a marriage of cloud based services with a stable web framework. However, more disruptive technology is already emerging. In particular, the key to truly scalable services has always been the approach to concurrency. A radical alternative to traditional threading model (embodied in Heroku) is emerging. In particular, successive attempts to solve the concurrency problem (discussed below) are converging towards a more radical approach; namely the so-called non-blocking option.

Client. The rise of the app store model is still taking shape. In particular, the introduction of this model to the general web through the Google Chrome Web Store and Social Networking Applications, may generate unforeseen consequences and trajectories in services and apps. For instance, the Chrome web store contains many applications that are indistinguishable from their apple app store equivalents. However, these applications are full HTML5 (not native), are by definition more cloud oriented, and are thus liberated from highly restrictive (and complex) native app development toolkits.

3 Concurrency Challenges

3.1 Threads

Diverse approaches to programmatically coping with concurrency have long been a source of contention among software developers. The evolution of the various approaches to concurrency is well illustrated in the C like languages, particularly Java. Although Java was designed with thread based concurrency in mind (unlike C and C++), its concurrency support has evolved significantly since its inception, with adjustments made to the core syntax, the libraries and the recommended approaches. The fundamental mechanism (synchronised keyword to serialise method access), has been supplemented with concurrent data structures, more expressive annotations, and an extensive rework of the concurrency model in Java 5 to incorporate a new executor framework. However, concurrent programming in Java is still regarded as complex and error prone, with non-determinism an ever present worry, even for systems long deployed in the field.

3.2 Actors

The java concurrency model is founded on the shared state semantics of a single multi-threaded process, whereby threads can share resources and memory, but with locks associated with specific data structures. Alternatives to this model have gained some ground. The actors model rules out any shared data structures (and their resource hungry locks), with concurrency achieved by message passing between autonomous threads - each thread (an actor) has exclusive access to its own data structures. In functional languages derived from Java (Scala,

Clojure), immutability itself is elevated to be the default programming model. This requires wholesale adoption of functional approaches (or object-functions hybrids in the case of Scala [1]), with the consequent profound change in programming style and heritage. With all of these approaches there is one common characteristic. Separate threads are created, with their own stacks and program counters. Although the opportunities for inter-thread synchronization vary, such synchronization must occur at some stage, with consequent overhead associated with task switching, memory usage and general processor load.

3.3 Non Blocking IO

There is an alternative, which has its origins in an era that predates the general acceptance of multi threaded infrastructure. Evolved to meet the requirements for responsive I/O in single processor systems, it sometimes takes the term Non Blocking IO, although this term has also been applied to threaded designs. Originally devised as a set of interrupts and associated daisy chained interrupt handlers, in the modern sense (if we can call it that), non-blocking I/O implies an extensive use of callbacks in API design and usage. In this context, all opportunities for blocking are replaced by passing a callback parameter, to be invoked on completion of the deferred task or I/O request. A somewhat counter-intuitive programming style, it has been criticised for its verbosity and general awkwardness. In certain programming languages it is indeed verbose - Java in particular is encumbered with a high-ceremony anonymous inner class syntax which makes callbacks quite difficult to orchestrate. Also, in Java and other languages of that generation, the callbacks are limited in scope and place severe restrictions around the context they can access. What they lack is a closure capability - essentially a form of delegate/callback/function handle - which also carries (encloses) a well defined context that can be safely accessed when it is activated. Closures have become a hot topic in programming language recently, and Java itself is slated to this capability in future versions. JVM derived languages such as Scala and Groovy have this capability, as does Closjure via its Lisp heritage. In fact the term closure originates from these functional languages.

3.4 C10K Problem

This challenge is made concrete by what is known as the C10K problem, first posed by Dan Kegel in 2003 [9]. The C10K is this: how can you service 10000 concurrent clients on one machine and other research has investigated the reverse of this problem [3]. The idea is that you have 10000 web browsers, or 10000 mobile phones all asking the same single machine to provide a bank balance or process an e-commerce transaction. Thats quite a heavy load. Java solves this by using threads, which are way to simulate parallel processing on a single physical machine. Threads have been the workhorse of high capacity web servers for the last ten years, and a technique known as thread pooling is considered to be industry best practice. However, threads are not suitable for high capacity servers. Each thread consumes memory and processing power, and theres only

so much of that to go round. Further threads introduce complex programming issues, including one known as deadlock. Deadlock happens when two threads wait for each other. They are both jammed and cannot move forward without the other releasing a hold on a particular resource. When this happens, the client is caught in the middle and waits, forever. The website, or cloud service, is effectively down.

Event Based Programming. There is a solution to this problem; event-based programming. Unlike threads, events are light-weight constructs. Instead of assigning resources in advance, the system triggers code to execute only when there is data available. This is much more efficient. It is a different style of programming, one that has not been quite as fashionable as threads. The event-based approach is well suited to the cost structure of modern computing, it is resource efficient, and enables one to build C10K-capable systems on cheap commodity hardware. Threads also lead to a style of programming that is known as synchronous blocking code. For example, when a thread has to get data from a database, it hangs around (blocks) waiting for the data to be returned. If multiple database queries have to run to build a web page (to get the users cart, and then the product details, and finally the current special offers), then these have to happen one after another, in other words in a synchronous fashion. You can see that this leads to a lot of threads alive at the same time in one machine, which eventually runs out of resources. The event based, non-blocking model is different. In this case, the code does not wait for the database. Instead it asks to be notified when the database responds, hence it is known as non-blocking code. Furthermore, multiple activities do not need to wait on each other, so the code can be asynchronous, and not one step after another (synchronous). This leads to highly efficient code that can meet the C10K challenge. JavaScript is uniquely suited to event-based programming because it was designed to handle events. Originally these events were mouse clicks, but now they can be database results. There is no difference at an architectural level inside the event loop, the place where events are doled out. As a result of its early design choices to solve a seemingly unrelated problem, JavaScript, as a language, turns out to be perfectly designed for building efficient services [2].

Node.js. The one missing piece of the JavaScript puzzle is a high performance implementation. Java overcame its early sloth, and was progressively optimised by Sun. JavaScript needed a serious corporate sponsor to really get the final raw performance boost that it needed. Google has stepped forward. Google needed fast JavaScript so that its services like Gmail and Google Calendar would work well and be fast for end-users. To do this, Google developed the V8 JavaScript engine [16], which compiles JavaScript into highly optimised machine code on the fly. Google open-sourced the V8 engine; and it has been adapted by the open source community for cloud computing. The cloud computing version of V8 is known as Node.js, a high performance JavaScript environment for servers [6] [7]. All the pieces are now in place. The industry momentum from cloud and

mobile computing, the conceptual movement towards event-based systems and the cultural movement towards accepting JavaScript as a serious language [10]. All these drive towards a tipping point that has begun to accelerate: JavaScript is the language of the next wave

4 Platforms and Frameworks

4.1 Platforms

Cloud Computing. Cloud computing is one of the key drivers compelling the current wave of innovation [5]. For the first time, corporations are moving their sensitive data and operations outside of the building. They are placing mission critical systems into the cloud. Cloud computing is now an abused term. It means everything and nothing. But one thing that it does mean is that computing capacity is now metered by usage. Technology challenges are not solved by sinking capital into big iron servers. Instead, the operating expense dominates, driving the need for highly efficient solutions. The momentum for green energy only exacerbates this trend.

Mobile Computing. Mobile computing represents the other side of the coin. The increasing capabilities of mobile devices drive a virtuous circle of cloud-based support services leading to better devices that access more of the cloud, leading to ever more cloud services. The problem with mobile devices is the severe fragmentation. Many different platforms, technologies and form factors vie for dominance, without a clear leader in all categories. The cost of supporting more than one or two platforms is prohibitive. And yet there is a quick and easy solution: the new HTML5 standard for web applications. This standard offers a range of new features such as offline apps and video and audio capabilities that give mobile web applications almost the same abilities as native device applications. As HTML5 adoption grows, more and more mobile application will be developed using HTML5, and of course, made interactive using JavaScript, the language of the web.

Social Applications. Social applications refers to a class of applications that integrate with one or more social networks, for example twitter, facebook, foursquare [21][22][23]. Social Applications are designed to be cross platform, accessible on mobile devices, potentially provisioned in the cloud or in standalone hardware and designed to emulate native applications. A social application can take the following forms:

- Within the context of a social networks primary site - usually within an iframe tag on the page such as a facebook canvas app [24]
- As a separate web site that integrates with a social network through an API
- As a mobile web app that integrates with a social network through an API
- Data Drivers for database and resource access

– As a native smartphone application that integrates with a social network through an API

Whilst this encompasses a broad range of possible applications, the common thread is that of socialization. The effect of socialization is that it can cause some applications to become very popular amongst a user base very quickly. This is often referred to as viral spread because the application is passed from one user to others in their social graph and then on to friends of friends. The imact of this viral spread on an appilcation is to cause sudden spikes in server load for a web application. Note that this also holds true for native mobile applications as most native apps require some back end server component. Some of the most popular social applications have quickly gained user bases in the many millions. Advertisers and large brands are becoming aware of the possibility of engaging their customers and potential customers through social networks like facebook. This can be done through custom social applications that typically provide some reward to a consumer in exchange for engaging with the application. The advertiser in question is looking to take advantage of viral spread in order to amplify its message across the social network. For a brand this can be a very cost effective means of advertising when compared to traditional print and TV advertising. The life cycle of a social application in the field is typically short lived. and will consist of an initial release and promotion phase to seed the application with a user base. The app may or may not then spread virally at that point. Typically the promotion will run for a short period of time - of the order of a few weeks. It will then come to the end of its life and usage will drop off as the next promotion is started. A more cost effective way of developing these types of applications both during development and in operational deployment was needed.

4.2 Frameworks

It is only a few short years since Ruby on Rails (RoR) was launched. At the time if its inception, RoR was a highly innovative development framework [13][14]. The key driver for mass adoption of RoR was hugely increased developer productivity through "convention over configuration", an approach which has now certainly entered the zeitgeist and which has been adopted by almost all of the current development stacks: for example Pythons Django or PHPs Cake. The predominant application deployment model for most organizations during the rise of RoR was owned server infrastructure: i.e. make some capital investment in server hardware on which to deploy applications. Under this model operational expenditure was relatively static and was based on monthly costs for colocation and bandwidth. Operational efficiency of deployed, in the field, applications was not so important for anyone but the really large sites, as long as the application could scale horizontally to some degree, capacity could be added by purchasing more hardware which was a one off hit if it could be accommodated into existing cabinets / racks. With the mass adoption of cloud computing, this model is flipped on its head. Deploying to the cloud requires little or no capital investment; however, operational expenditure is now directly tied to the efficiency of

deployed applications. There is now a clear economic driver for efficient web applications and services. Whilst advances have been made by the major languages and frameworks, fundamentally, they do not make the best use of the available compute resources and are therefore not best suited to operation in the cloud. Furthermore experience has shown that these frameworks suffer from a number of other deficiencies:

SPA Disjunction. RoR type frameworks exhibit a Model View Controller (MVC) architectural structure. Under this paradigm, an application consists of a set of MVC triplets that are processed server side to render HTML back to the client. However, most modern applications no longer fit this model and are increasingly adopting the Single Page Application (SPA) or Multi-Single Page Application (MSPA) architectural style. Under this model static html is sent down to the client and acts as a basic application frame. Client side JavaScript then makes AJAX requests to hang the front end functional elements onto the application frame. Consequently much more of the application logic is implemented on the client in JavaScript. Whilst some work has been done in this area, notably backbone,js and Faux, none of the major MVC frameworks provide any governance or organizational structure for client side JavaScript. Developers are left to construct their own ad-hoc client side application architectures over libraries such as JQuery. This often means that the client can quickly degenerate to spaghetti code with little or no unit testing.

Code Duplication. The SPA Disjunction also leads to code duplication. Take for example the task of verifying and sanitizing user input. To provide a good user experience and rapid response time, this task is best done on the client. However for security reasons it must also be checked server side. Therefore this logic is typically implemented twice, once in javascript on the client and again on the server in the language appropriate to the framework being used (i.e. Ruby and Python).

Relational Database Assumption. RoR type frameworks were built around the assumption that the framework would talk to a single relational database. Indeed in early versions of rails it was difficult to introduce an additional relational data store into an application. All the current production frameworks make the unstated assumption that the back end is an SQL compliant database.

Language Proliferation. To work end to end with any current MVC framework one must be proficient in a minimum of five languages - SQL, one of Ruby/Python/PHP..., JavaScript, HTML5 and CSS. This can have one of two effects, either individual developers must context switch between the various languages depending on where they are in the stack at a given point in time, or a team is broken down into front end and back end specialists, a division which can cause delay and communication overhead when implementing application functionality.

5 Project Zeppelin

Zeppelin is a lightweight end to end javascript application framework based on node.js and designed to address some of the short comings outlined in the previous section. The mental disconnect from a developers point of view is removed using this approach. With the client authored in Javascript, the server side also speaking Javascript and the data store operating a noSQL design with Javascript bindings, the requirements from a developers point of view drops dramatically. The language proficiency is minimised to a trinity of Javascript, HTML5 and CSS, thus empowering the developer to manage and maintain all aspects of a system. A side effect of end to end Javascript is a low friction environment as technologies are not battling against each other. Zeppelin is designed to run many applications on the same stack instance. It can be considered a cross between a framework and an application container / server. The current beta software is based on the Architecture visible in Figure 1.

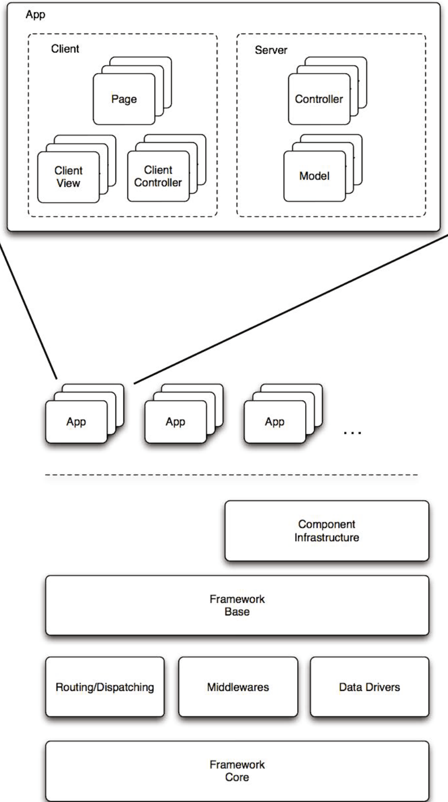


Fig. 1. Zeppelin Overview

Zeppelin consists of several core components including:

- Framework Core: A utility for setting up and tearing down applications. Stack configuration and management is also housed within the core.
- Routing/Dispatching: matches contollers to RISON urls for example this will match the url http://host/app/(controller.method)/params to a specific controller and method invocation. There is no routing table, all routing is handled dynamically
- Middlewares: global stack objects that provide services to applications. For example, authentication and authorization, logging
- Data Drivers for database and resource access
- Framework based implementations and utility functions for server side controllers and models.
- Component infrastructure: experimental shared components for use across multiple apps

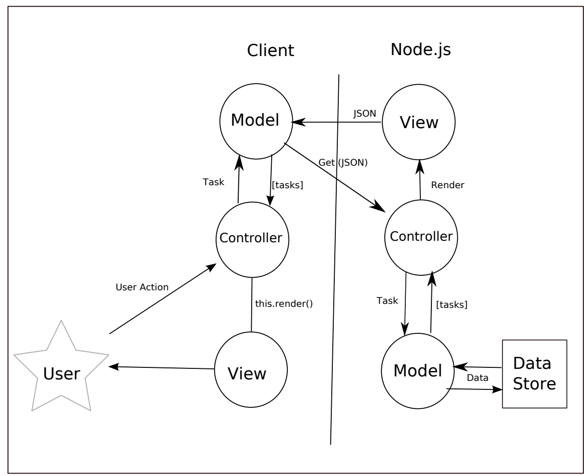


Fig. 2. Distributed MVC Architecture

The distributed MVC architecture, as seen in Figure 2 is a change from the traditional MVC Pattern. It retains the isolation of domain logic from the user interface but allows the pattern to function in a more scalable manner, with cloud style applications in mind. A shift in processing means the client device can take a larger responsibility allowing the server more room for receiving and processing requests, passively boosting scalability. This style also lends itself to code reuse and the rapid development and deployment of applications built on the stack.

Each Zeppelin application implements a distributed Model View Controller architecture comprising of:

- Client views: html fragments + templates
- Client Controllers: javascript objects implementing render() and bind()
- Client Models: Provide interface to server components of local storage
- Server Controllers: Server side entry points
- Server Models: Use data drivers to access file system, databases etc

A sequence diagram, visible in Figure 3, shows the communication flows between client and server during a typical application setup. Figure 4 shows the same client server components for a single application.

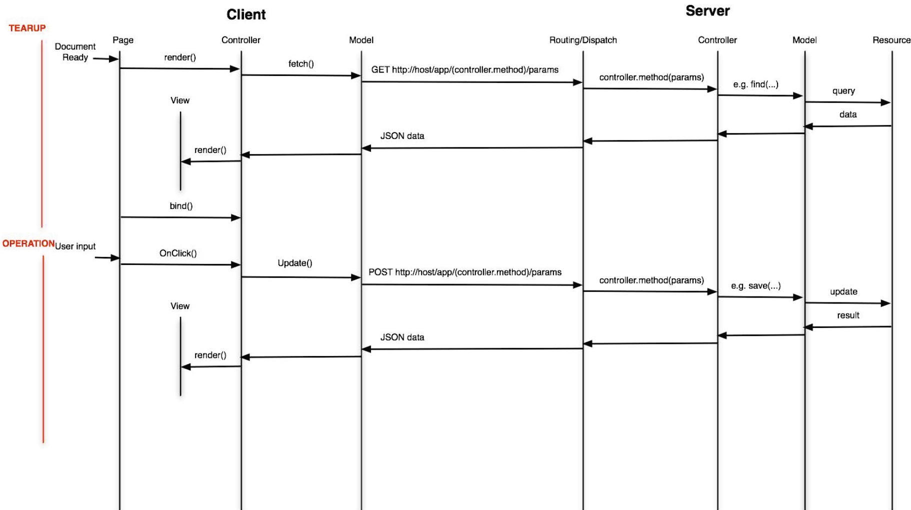


Fig. 3. Client Server Sequence Diagram

Zeppelin in the Field. Zeppelin has been used to successfully build a number of Facebook Applications including Ireland Town [18], developed by Betapond [17]. Ireland Town is a typical Facebook application, showing incredible growth and usage over a short period of time. Figure 5 shows some usage statistics for Users and User generated content (sharing on Facebook) over a three week period. From the statistics gathered, approximately 300,000 shared pieces of content, typically wall posts, were generated within a 21 day period. To highlight the potential for a viral application to reach millions of end users, consider this simple scenario. If each person who shared this content has typically 25 people in their network, that is 7,500,000 views on the primary message appearing on news feeds. If 10% of these interact with the message a further 18,750,000 views are generated. The application had the potential to have 26,250,000 total views on the message distributed from 300,000 initial interactions.

From a marketing point of view, this opens up a very large demographic through the sharing mechanism in a relatively short time scale. From an applications point of view, a fraction of 1% of 26 million people using the application

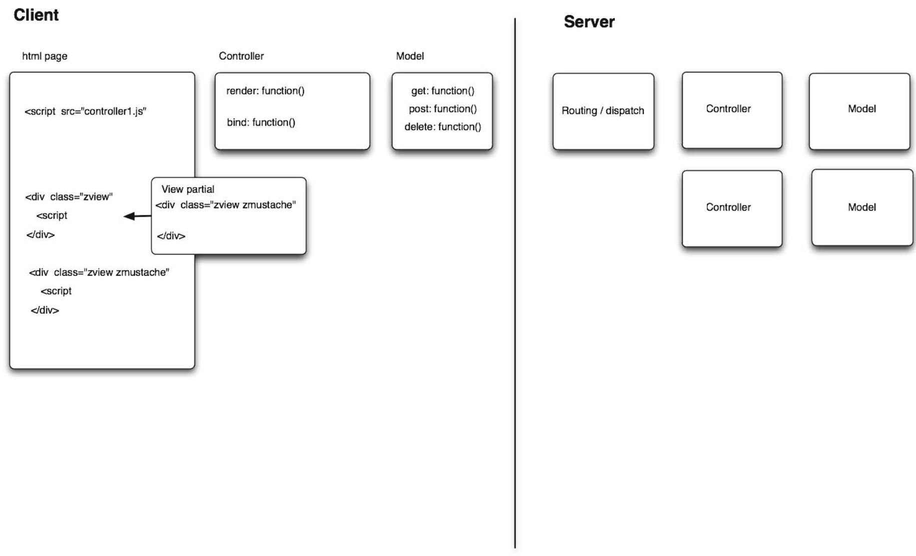


Fig. 4. Client Server Architecture

Fig. 5. Ireland Town Stats

represents a scalability conundrum that existing technology struggles with and interactions that the C10K problem did not foresee. The scenario presented is deliberately under provisioned. The average number of connections a person has is statistically placed at 120 by Facebook [25]. An application that crosses demographics and reaches a truly international audience will rapidly reach the usage of the Ireland Town application within a few hours of launch. The Ireland Town

application was thematically based around St. Patricks Day, limiting the global audience somewhat, but still producing a challenge scalability wise which helped test and shape the Zeppelin platform. As viral marketing techniques focus in on heavier application interaction, the potential to cripple an application and potentially halt a marketing campaign is imminent. Such a scenario is driving the design of Zeppelin and ensuring that applications are engineered to meet the challenges of cloud computing and social networking.

6 Future Work and Conclusion

The software developed as part of this work is freely available on [27] and is currently labelled as beta software. The project has another nine months of funded development with a clear strategy outlined for future work. Unit test and integration testing support is mandatory in any candidate software release and Zeppelin has identified a strategy for integrating this element. Expresso [11] will be used as a server side testing framework. A complimentary application testing suite, as realised by a Selenium Webrunner [12] for end to end testing of applications is also proposed for integration. Testing frameworks are constantly evolving and to that end, extensible bindings are made available, allowing end users of Zeppelin to integrate their testing framework of choice into the framework. Security and Privacy aspects of the framework need further refinement in order to bring the application in line with industry standards. Currently, standard attack tools such as Nessus [26] are used to test the framework for exploits allowing a swift response to vulnerabilities. The EternalS [19] project has a strong emphasis on long living socio technical systems, particularly focusing on engineering principles and architectural decisions. Outputs derived from EternalS are helping to shape the evolution of the Zeppelin Architecture. One such analysis surrounding Product Line engineering principles within social networks, will allow Zeppelin rapidly produce and deploy high quality applications. This paper examined the change in application platforms and put forward a case for a new paradigm shift, one which is a lightweight evolution of existing principles. The evolution, as we have described, is driven by changes in the wider web, namely cloud computing, and by how users interact with services. Predominantly mobile access of services provisioned through social network applications brings with it scalability challenges not previously encountered, or catered for in the design of existing application frameworks. We have put forward a Javascript based application framework, dubbed Zeppelin, and shown the environment in which it living.

Acknowledgments. This work has been co-financed by the European Commission - IST EternalS (FP7-247758) and the Enterprise Ireland Innovation Partnership Programme.

References

1. Oliveira, B., Gibbons, J.: Scala for Generic Programmers. In: Proceedings of the ACM SIGPLAN Workshop on Generic Programming (2008)

2. Griffin, L., Ryan, K., de Leastar, E., Botvich, B.: Scaling Instant Messaging Communication Services: A Comparison of Blocking and Non-Blocking techniques. In: IEEE International Symposium on Computers and Communications (2011)
3. Liu, D., Deters, R.: The Reverse C10K Problem for Server-Side Mashups. In: Feuerlicht, G., Lamersdorf, W. (eds.) ICSOC 2008. LNCS, vol. 5472, pp. 166–177. Springer, Heidelberg (2009)
4. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-Oriented Computing: State of the Art and Research Challenges. Computer 40(11), 38–45 (2007)
5. Liang, H., Chen, W., Shi, K.: Cloud computing: programming model and information exchange mechanism. In: Proceedings of the 2011 International Conference on Innovative Computing and Cloud Computing, ICCC 2011, pp. 10–12. ACM, New York (2011)
6. Node.js, http://nodejs.org/
7. Lerner, R.M.: At the forge: Node.JS, Linux J. (2011)
8. O'Reilly, T.: What is Web 2.0, http://oreilly.com/web2/archive/what-is-web-20.html
9. Kegel, D.: The C10k problem, http://www.kegel.com/c10k.html
10. Tilkov, S., Vinoski, S.: Node.js: Using Javascript to Build High-Performance Network Programs. IEEE Internet Computing (2010)
11. Expresso Test Framework, http://visionmedia.github.com/expresso/
12. Selenium Junit Web Runner, http://code.google.com/p/selenium-junit-web-runner/
13. Maximilien, E.M.: Web Services on Rails: Using Ruby and Rails for Web Services Development and Mashups. In: IEEE International Conference on Services Computing (2006)
14. Viswanathan, V.: Rapid Web Application Development: A Ruby on Rails Tutorial. IEEE Software 25(6), 98–106 (2008)
15. Cattell, R.: Scalable SQL and NoSQL data stores. SIGMOD Rec. 39, 12–27 (2011)
16. Google V8 Javascript Engine, http://code.google.com/p/v8/
17. Betapond, http://betapond.com/
18. Ireland Town, http://apps.facebook.com/irelandtown/town
19. EternalS, https://www.eternals.eu/
20. Long, B., Long, B.W.: Formal specification of Java concurrency to assist software verification. In: Parallel and Distributed Processing Symposium (2003)
21. Twitter, http://twitter.com/
22. Facebook, http://facebook.com/
23. FourSquare, http://foursquare.com/
24. Facebook Canvas Specification, http://developers.facebook.com/docs/guides/canvas/
25. Facebook Statistics, http://www.facebook.com/press/info.php?statistics
26. Nessus, http://www.tenable.com/products/nessus
27. Zeppelin Code, https://github.com/pelger/Zeppelin

Managing Adaptivity in Parallel Systems

Marco Aldinucci[2,⋆], Marco Danelutto[1], Peter Kilpatrick[3],
Carlo Montangero[1], and Laura Semini[1]

[1] Dept. of Computer Science, Univ. of Pisa
[2] Dept. of Computer Science, Univ. of Torino
[3] Dept. of Computer Science, Queen's Univ. of Belfast
aldinuc@di.unito.it, p.kilpatrick@qub.ac.uk,
{marcod,monta,semini}@di.unipi.it

Abstract. The management of non-functional features (performance, security, power management, etc.) is traditionally a difficult, error prone task for programmers of parallel applications. To take care of these non-functional features, autonomic managers running policies represented as rules using sensors and actuators to monitor and transform a running parallel application may be used. We discuss an approach aimed at providing formal tool support to the integration of independently developed autonomic managers taking care of different non-functional concerns within the same parallel application. Our approach builds on the Behavioural Skeleton experience (autonomic management of non-functional features in structured parallel applications) and on previous results on conflict detection and resolution in rule-based systems.

1 Introduction

When designing, implementing and debugging parallel applications a number of non-functional concerns typically have to be taken into account and properly managed. A *non-functional* concern (sometimes referred to as *extra* functional concern and more recently referred to as *quality attribute*) is a feature not directly affecting *what* the parallel application computes, that is the parallel application result. Rather, it is a feature affecting *how* the parallel application result is computed. Notable examples of non-functional concerns in parallel applications are performance, fault tolerance, security, power management, with performance often being the most important.

Properly managing a non-functional concern usually requires the design, implementation and tuning of code additional to that needed to compute the results of a parallel application (the so-called *business code*). The kind of code needed to manage a non-functional concern poses additional requirements on the application programmer, as correct management of non-functional concerns usually requires a quite deep understanding of the target architecture, which is not usually required when writing business code (only). As an example, performance

⋆ This work has been partially supported by EU FP7 grant IST-2011-288570 "ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems".

B. Beckert et al. (Eds.): FMCO 2011, LNCS 7542, pp. 199–217, 2013.
© Springer-Verlag Berlin Heidelberg 2013

optimization requires a clear vision of target architecture features in order to be effective. Moreover, the non-functional code is often deeply interwoven with the business logic code, thus resulting in much more difficult debugging and tuning of both business logic and non-functional concern management code.

Radically different approaches may be taken to manage non-functional concerns if we recognize that non-functional concern management is a completely independent activity w.r.t. business logic (functional code) development. In fact, non-functional concern management can be organized as a policy insurance procedure piggy backed onto business logic code. The policies used while managing non-functional concerns are the *non-functional programs* and the mechanisms used to implement these programs–typically those mechanisms used to "sense" the computation status and to "actuate" policy decisions–represent the *assembler instructions* of non-functional management.

If this perspective is taken, then management of non-functional concerns may be implemented as an autonomic engine associated to the business logic code. We can implement MAPE (monitor, analyze, plan, execute) loop based managers where monitoring and execution of actions–those devised by policies in the analyze phase and planned by other policies in the planning phase–happen through the sensor and actuator mechanisms provided by the non-functional concern management assembly instructions. Fig. 1 outlines this general idea.

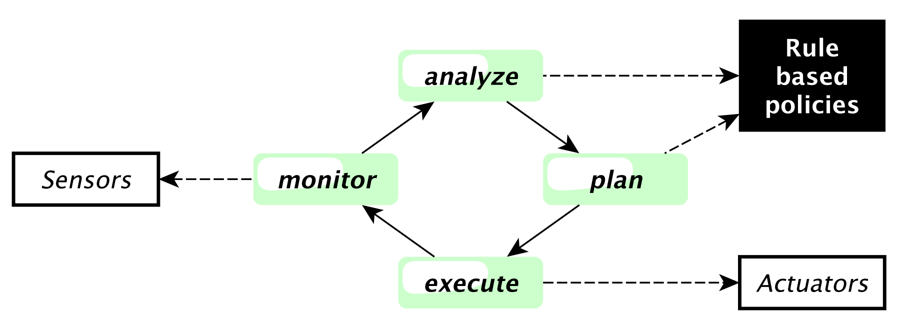


Fig. 1. MAPE loop in autonomic management: the control flow is represented by solid arrow lines and the dependencies are represented by dashed lines

Previous work has demonstrated the feasibility of such an approach to non-functional concern management [2,3,18,1]. These works also pointed out two interesting and somehow conflicting facts:

- Best management policies may be provided by experts in the specific non-functional concern, rather than by "general purpose" non-functional concern experts and/or application programmers.
- Different non-functional concern management policies may lead to conflicts, that is decisions in relation to management of non-functional concern A may impair decisions in relation to management of non-functional concern B.

Therefore autonomic management of different non-functional concerns poses an interesting problem: how can we "merge" policies managing different non-functional concerns without incurring serious penalties due to policy conflicts?

The rest of the paper introduces non-functional concerns and their autonomic management in more detail (Sec. 2 and Sec. 3). Then conflict detection techniques are introduced (Sec. 4). Sec. 5 discusses formal support for policy merging and presents experimental results to assess the complete methodology. Finally, conclusions are drawn in Sec. 6.

2 Non-functional Concern Management in Parallel Computing

As stated in Sec. 1, a non-functional concern is a feature related to *how* the results of an application are computed rather than to *what* these results actually are. Typical non-functional concerns in parallel applications include:

Performance. By far, the most significant non-functional concern in parallel programming. Usually, two distinct kinds of optimization may be required, either latency or service time optimization, with differing implications for the pattern used to exploit parallelism.

Security. Security requirements may be related to data processed and/or to the code used to process input data to get output results. These requirements may vary depending on the kind of resources used to compute the parallel application: shared, private, reserved (i.e. not private, but with exclusive access guaranteed).

Fault Tolerance. Considering the number of resources involved in large-scale parallel applications, it is quite common to experience hardware faults during the execution of an application. Thus fault tolerance is particularly critical to ensure correct completion of applications in the event of failure of (part of) the resources used, especially in the case of long-running applications.

Power Management. If different resources are available (with differences both in terms of power consumption and of performance delivered) power saving becomes a fundamental option in parallel processing, especially at large/extreme scale.

In most cases, the management of these non-functional concerns requires quite complex activities, including:

- Adoption of more complex mechanisms and tools with respect to those needed to support business code only. For example, to ensure security, SSL connections may be required instead of plain TCP/IP connections.
- Parallelization of sequential code or further parallelization of parallel code. For example, in a data parallel computation the input data should be partitioned among a larger number of threads to ensure a shortened completion time of the application. Or the presence of a sequential bottleneck in a parallel computation may require parallelization of the bottleneck code.

– Complete restructuring of the parallel application, i.e. changing the parallel design pattern used to exploit parallelism in the application. For example, having first used a stream parallel pattern, we may realize that the performance of our parallel application is not sufficient and may therefore apply some data parallel pattern also on the different stream parallel pattern components.

In general, various policies may be adopted to deal with non-functional concerns, with different applicability pre-conditions and different results. For example, when dealing with performance, if an application is not performing as expected when running on a heterogeneous architecture, we can either try to move parallel computation components of the application to more powerful architecture nodes (processing elements) or we can try to improve the "structure" of the parallel application (e.g. by changing the parallel pattern used) to give better performance on the existing and available computing resources.

It is worth pointing out that, in general, it is easier to devise suitable management policies when the structure of the parallel application is completely exposed. If the structure is not exposed, it is much more difficult to determine what exactly is going on and thus to plan corrective action in the event of a (non-functional) malfunction of the application. Indeed, without a general view of the application parallel structure, it may even be difficult to realize that there is a non-functional malfunction.

If the parallel pattern of the application at hand is completely exposed we are enabled:

1. to verify whether the application is performing as expected, as the (parallel) design pattern used will come with models that can be verified while the application is running; and
2. to take the decisions suggested by the design pattern used to correct possible problems/malfunctions.

Of course, the parallel pattern–or the pattern composition–used within the application may be identified in two distinct ways: i) by analyzing the HLL (High Level Language) code used to program the application (e.g. where we use a programming framework based on algorithmic skeletons), or ii) by running some kind of (data flow) analysis on the application code to determine whether the underlying parallel activities fit one of the known parallel patterns.

3 Autonomic Management of Non-functional Concerns

Autonomic managers of non-functional concerns may be programmed as outlined in Sec. 1 using MAPE loops. A MAPE loop is a control loop cycling on four different phases (see Fig. 1):

M A *monitoring* phase, where the current status of the running parallel application is observed by collecting data on what happens on the actual target architecture: how many (partial) results have been computed, the time spent

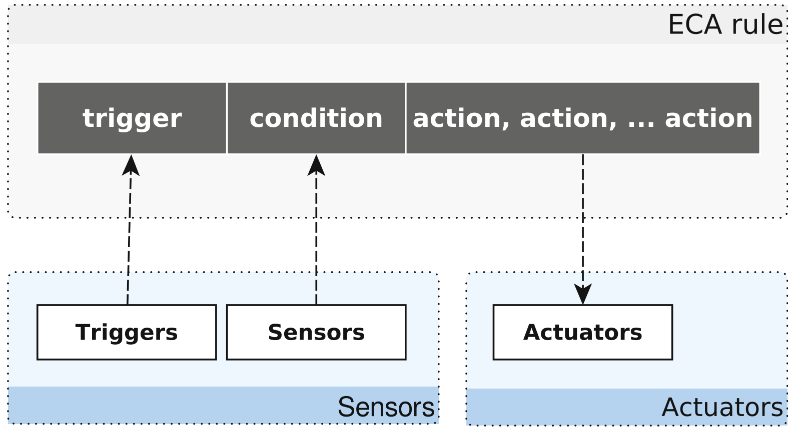


Fig. 2. Implementation of ECA rules

in the different running tasks, the amount of resources used (CPU, Memory, Network), etc.

A An *analyze* phase, where the current situation is analyzed, the behaviour of the parallel application is compared to the expected behaviour and, possibly, a plan to improve application behaviour is selected.

P A *plan* phase, where the decisions taken in the analyze phase are turned into a sequence of actions to be run on the current application.

E An *execute* application, where the plan is actually executed.

The monitoring and plan+execute phases rely on the existence of a set of mechanisms with the ability to "sense" application behaviour and to "act upon" application execution, i.e. to apply the plans devised by the manager policies in the analyze+plan phases. These mechanisms–sensors and actuators–represent "passive" code, as they are just called from within the manager. They also represent *de facto* the interface of the autonomic manager with the (running) business code of the application and determine the kind of policies that can be effectively implemented in the manager.

To clarify the concept, consider an application whose parallel pattern is based on the master/worker paradigm. The availability of sensors reporting to the monitoring phase the number of workers executing and the service time delivered by the master/worker combination determines the capability to react to poorly performing application states. In the same way, the availability of sensors capable of reporting whether a parallel application component is running on a private or on a public resource will enable the manager policies to take correct decisions to ensure application security. On the other hand, the existence of mechanisms

(actuators) capable of stopping and restarting the application, recruiting new resources and deploying and starting active code on the these newly acquired nodes is fundamental to implementation of smart management policies, such as increasing the number of workers in the master/worker pattern or moving an application component from a public node to a private/reserved node.

As far as the "active" part of the MAPE loop is concerned–the analyze and plan phases–different choices can be made. Plain code can be used to hard-wire policies and plans and to call the sensor and actuator mechanisms. However, if we wish to experiment with different policies, or investigate changing policies "on-the-fly" depending on the perceived application status, a more dynamic and declarative solution is necessary. Various systems, including the authors' Behavioural skeletons [2,3], use ECA (Event Condition Action) rule systems to implement manager policies.

An ECA rule is applied in a context that consists of the status of the system at the beginning of the MAPE cycle and the set of events that occurred in the previous cycle. Events may be *external*, that is generated in the system environment, or *internal*, that is generated as part of the effect of an action performed in the previous cycle. The application of a rule results in a new state and possibly in (internal) events, to be considered in the next cycle, when also the external events received in the current cycle will be considered.

More precisely, an ECA rule is a triple

$$\langle trigger, condition, action \rangle$$

Whether a rule is applied in a MAPE cycle depends on its trigger and condition. The trigger is a pattern describing of the events that may cause the application of the rule (a.k.a. *firing* of the rule). At the beginning of the cycle, the trigger is matched with the events in the current context. In case of success, the rule becomes ready to fire: it actually fires, that is, its action is executed, only if its condition holds in the current state. An event that is not matched, or is matched when the condition does not hold, is lost. A single event can enable two or more rules if it matches their triggers. In the case two or more rules are enabled in an evaluation step they are fired concurrently.

The matching process may bind parameters in the trigger to the values carried by the event: the scope of such binding covers the rule condition and action, thus enhancing the capability of the notation to express complex policies. For the same purpose, two triggers may be disjoint, meaning that either is sufficient to apply a rule, and conditions may be combined with the standard logical operators.

Fig. 2 shows how selectors and actuators can enact ECA rules.

The use of ECA rules allows a better implementation of the manager policies. In particular, we use the triggering event to start rule evaluation. In previous work, we used JBoss rule syntax to express management rules. In that case rules were tested cyclically for fireability. The period of the cycle *de facto* determined the MAPE loop efficiency, as "too slow" loops react poorly and "too fast" loops may lead to overly rapid decisions. In the following sections, we adopt the rule system of Appel (see Sect. 5.1) to implement our rule-based manager programs.

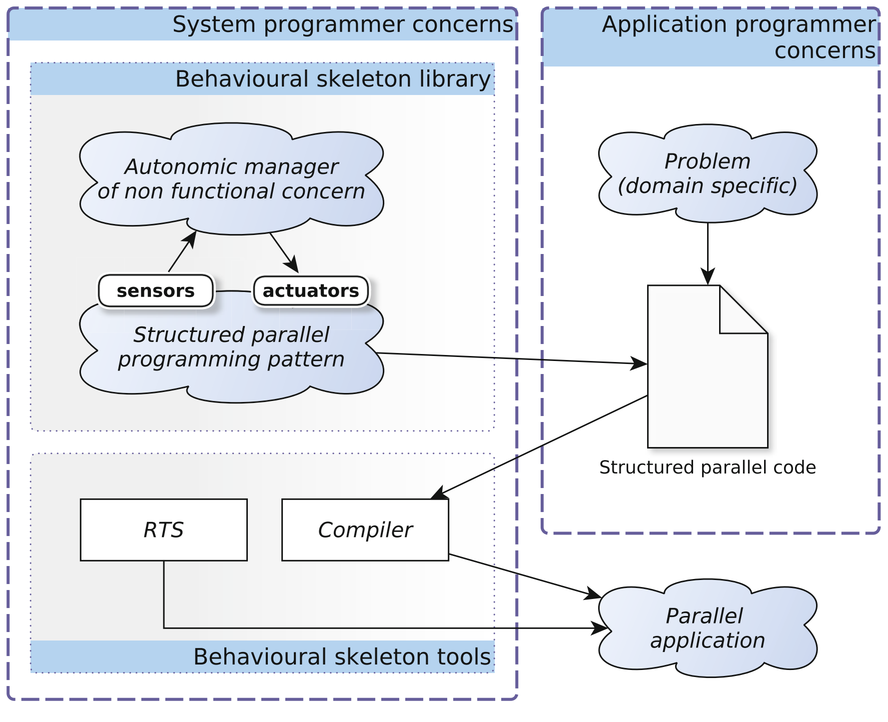


Fig. 3. Behavioural skeletons

Appel chooses rules for firing using a loop such as that mentioned above. However, the trigger events are gathered continually and an ordered list of events is exposed to the rule system at each loop iteration.

It's worth pointing out here that "planning" activities in our MAPE loop are not actually *proper* planning activities. Rather, the "plan" step in the loop consists in applying a plan that has been already coded in the action part of the ECA rules used as the program of the autonomic manager. These rules may also include an action part that somehow modifies the rule set. For example, a rule priority may be lowered or a rule may be substituted by a different one which more precisely reflects the actions needed in the current situation. This notwithstanding, the "plan" phase is actually a kind of "actuate one of the already established plans" phase. At a rather higher level of abstraction, the process leading to the design of the rules used as the program of the MAPE loop is a kind of MAPE loop itself. The current situation is monitored and then it is analyzed. During the analysis phase a policy is eventually identified which turns into a plan to be actuated/executed by generating suitable loops for our run time MAPE loop.

3.1 Behavioural Skeletons

Building on the concepts detailed in the previous Sections, we proposed some time ago the concept of *Behavioural skeleton*, i.e. of a parallel design pattern coupled with an autonomic manager taking care of a non-functional concern. In the original behavioural skeleton design, the parallel design patterns considered were the traditional ones in stream parallel computing models, that is *task farm* and *pipeline*. Task farm (a.k.a. abstraction of the master/worker implementation pattern) completely captures and models embarrassingly parallel computation on streams. Pipeline, instead, captures and models computation in stages, without backward communications. Also, the original behavioural skeleton design considered management of only a single non-functional concern: performance.

The behavioural skeleton approach is outlined in Fig. 3. A behavioural skeleton library is made available to the application programmer. The library contains several composable behavioural skeletons. Each behavioural skeleton consists of a parallel design pattern *and* of an autonomic manager running a MAPE loop and using an ECA rule system to implement policies. Suitable sensors and actuators are implemented within the parallel design pattern implementation to support autonomic manager activities.

The application programmer in charge of writing a parallel application chooses a behavioural skeleton or some composition of behavioural skeletons from the BS library and provides the behavioural skeleton(s) business logic parameters. For example, if the pattern used to express parallelism within the application is a pipeline, the application programmer chooses the pipeline behavioural skeleton and instantiates it passing as parameters the code (wrappers) implementing the business logic of the pipeline stages. If one of the stages has to be further parallelized, the application programmer may pass as pipeline stage an instance of a task farm whose worker parameter implements the parallel stage business logic.

Once the application programmer has written his/her application using behavioural skeletons, a compiler takes care of producing suitable parallel code for the target architecture at hand. This code relies on the functions provided by the behavioural skeleton run time library, of course.

It is worth pointing out several notable features of this approach:

- the system concerns (those requiring specific knowledge concerning the target architecture/system at hand) and the application concerns (those requiring more domain specific knowledge related to the application field) are kept completely separate. Separation of concerns clearly enforces productivity and efficiency in both application and system programmer activities.
- the application time-to-deploy is significantly reduced by reuse of behavioural skeleton library components.
- performance portability across different architectures is the responsibility of system programmers (as opposed to application programmers) who provide, in the behavioural skeleton library, components specific for the different target architectures.

– policy programmability is ensured by the ECA rule system embedded in the autonomic manager MAPE loop. Programming rules (declarative style) is much more user friendly and efficient than writing specific code using the sensors and actuators provided by the associated design pattern.

Leveraging on all these attractive properties, a prototype implementation of behavioural skeletons on top of the ProActive/GCM middleware has been demonstrated to be able to carefully manage performance in stream parallel applications [3].

4 Conflict Detection and Resolution in Rule-Based Systems

Policy conflict has been recognized as a problem and there have been some attempts to address it, mostly in the domain of access or resource control [17].

Kind of Conflicts. ECA rules conflict if (1) they may be triggered at the same time and (2) their conditions overlap and (3) their actions conflict. While this definition makes complete sense only in a specific application domain, as one must be aware of what conflicting actions are, the problem is inherent to policies. To ensure that policies can be applied it is necessary to remove conflicts. This process involves two stages: first one needs to identify whether conflicts can occur, that is to *detect* conflicts, and then to remove them, that is *resolve* conflicts.

The general definition of policy conflict can be extended to accommodate some special cases.

In [8] the authors discuss what it means for two rules to be triggered at the same time, providing two different interpretations: their trigger sets overlap (and the actual triggering event is in the overlap) or the action of one is in the trigger set of the other. The former case has been called STI (Shared Trigger Interaction) and the latter SAI (Sequential Action Interaction).

More generally, and this provides interesting future work in our case, the designer may be interested in specifying conflicts on the basis of traces, i.e. define as conflicting rules that (1) may be applied within n MAPE loops and (2) whose actions conflict.

One further aspect to consider, and this is again based on experience in feature interaction, is the question as to how many policies are required to generate a conflict. In the community, discussions have taken place around a topic called “three-way interaction”. In the feature interaction detection contest at FIW2000[9] this was an issue, and the community decided that there are two types of three-way interaction: those where there is already an interaction between one or more pairs of the three features and those where the interaction only exists if the triple is present. The latter were termed “true” three-way interactions.

Nothing has been written about true three-way interaction, as only one, quite contrived, example of such an interaction has been found. Hence we consider realistic to assume that no "true" three-way interaction may occur, and limit ourselves to pair wise checking.

Conflict Detection Time. We distinguish between design time (static) and run-time detection. In run-time detection, conflicts, if any, are looked for at each execution step among the rules that can be applied at the step. When conflict detection is anticipated at design-time, rules are filtered before being entered in the policy engine, to detect those that would originate conflicts. In this way we can provide the user with confidence that the rules are conflict free.

In our former works we addressed design time detection. In [13,14], we take a logic–based approach to this end: conflicts are detected by deducing specific formulae in a suitable temporal logic theory. In [6] we exploit the use of model checking to detect policy conflict. This is the approach employed in this paper.

Layouni et al. in [11] also experimented with the use of the model checker Alloy [5] to support policy conflict resolution.

Conflict Resolution. Conflict resolution can in general be attempted in a number of ways, and which is best suited depends on the situation. We can broadly distinguish between resolution at design–time and resolution at run–time. The taxonomy of policy conflict in [17] makes explicit that design-time resolution is always feasible when policies are co-located and owned by the same user. In this case resolution will be a redesign of the policies. However, when policies are distributed, this is not always possible and it is preferable to deal with conflicts at run-time. Resolution in this case may exploit priorities among policies, activating only policies with greater precedence. Nevertheless, there is a wide spectrum between the two extremes of co-location and complete distributed placement, and any conflict that is resolved before run-time is of benefit.

A comprehensive survey on detection and resolution techniques in three well-known policy management approaches, KAoS, Rei and Ponder, is found in [20].

5 Multiple Non-functional Concern Management: Formal Tool Support

The ability to develop independent managers and to modify them to accomplish coordinated management of multiple concerns is attractive for two reasons: it enforces modular design and reuse; and allows better use of domain specific knowledge of different non-functional concerns.

However, combining a set of single-concern managers may be difficult to achieve since it requires expertise in *all* of the non-functional concerns to be coordinated, and because the sheer number of evolution paths of the combined managers may make it extremely difficult for the human to identify the possibility of a conflict arising.

Model checking tools may provide fundamental support, however, as proposed in [6,10]: "conflicts" can be *detected* by model checking, once the conflicting atomic actions have been identified. More precisely, the whole design phase includes the following steps:

– Independent experts design and implement policies relative to distinct non-functional concerns.
– A set of conflicting actions is defined, such that a pair of actions a_i, a_j are in the set *iff* action a_i "undoes" action a_j and vice versa.
– A formal model of the rule system is derived, which is fed to a model checker.
– The model checker is used to check formulas stating that conflicting actions may occur "at the same time", that is in the same MAPE loop iteration.
– The traces leading to the situation with the conflicting actions obtained from the model checker are used to change the rules to handle conflicts.[1]

5.1 An Experiment in Static Conflict Detection for Autonomic Managers

In the sequel, we first describe our experimental setting, and then discuss our first results, with respect to the likelihood of applying the technique to real life examples. The ingredients of the technique are a policy language, a model checker and a translator able to generate a checkable model from the policies.

Appel. We use Appel [22,21] to write the management rules. Appel is a general language for expressing policies in a variety of application domains: it is conceived with a clear separation between the *core* language and its specialization for concrete *domains*, a separation which turns out very useful for our purposes.

In Appel a *policy* consists of a number of *policy rules*, grouped using a number of operators (**seq**uential, **par**allel, **g**uarded and **u**nguarded choice).

A policy rule has the following syntax:

[**when** *trigger*] [**if** *condition*] **do** *action*

The core language defines the structure but not the details of these parts, which are specifically defined for each application domain: base triggers and actions are domain-specific atoms; an atomic condition is either a domain-specific or a more generic (e.g. time) predicate. This allows the core language to be used for different purposes. In our case, as mentioned above, the triggers relate to active sensors, conditions to (passive) sensors, and actions to actuators.

Triggers can be combined with a disjunction, complex conditions can be built with Boolean operators, and a few operators (**and**, **andthen**, **or** and **orelse**) are available to create composite actions.

[1] At the moment conflicts are identified by the model checker, but then the actions needed to resolve the situation (i.e. the modifications to the manager rules) are performed by humans. The asymptote is to have this part also executed automatically.

The semantics of APPEL [,] which before was only defined informally, as with most policy languages, has been formally defined by translation into the temporal logic ΔDSTL(x) [15,16].

Though APPEL supports also a notion of priority among the rules, we do not exploit this currently.

UMC. This is an on-the-fly analysis framework [12,23,19] that allows the user

1. to interactively explore the behaviour of a UML state machine;
2. to visualize abstract slices of its behaviour; and
3. to perform local model checking of UCTL formulae, UCTL being a branching-time temporal logic [7].

The last feature is the most important for our purposes, but the previous ones are very useful once a conflict is detected and we need a deep understanding of what is happening to resolve it.

UCTL allows specification of the properties that a state should satisfy and combination of these basic predicates with advanced temporal operators dealing also with the performed actions. Some care must be taken in writing the formulae that characterize the conflicts to be detected, since they are checked not against the traces of the UML state machine, but against the traces of an equivalent standard state machine – generated by UMC – where parallelism is resolved with interleaving. So to detect two conflicting parallel actions one has to detect any sequence of the actions in any path in the traces.

Appel2UMC. We have defined a semantics-preserving compositional mapping from APPEL to UML, suitable for model checking with UMC. Since UMC operates on UML state machines, the target of the mapping happens to be a subset of UML state machines: policies and policy groups are defined using composite states, i.e. states with structure reflecting the one imposed by the APPEL operators onto policies and actions.

To derive a UML state machine model of the system to feed the checker, we follow the approach of [6]: APPEL policies are automatically mapped to a UMC specification, i.e. the description of a UML state machine, in the UMC textual input format.

The mapping is based on the APPEL semantics given in terms of UML state machines. Actually, the mapping needs not consider the actual semantics of the actions, but only an abstract one, where an action may result in a *success* or a *failure*. Intuitively, these notions entail that an action may complete normally (success) or may abort for some reason (failure), and APPEL leaves the specifics of an action success or failure to the domain. However, it defines the success or failure of a composed action as a composition of the successes and failures of the actions under composition. Therefore, for the translation, actions can be treated as propositional atoms.

The prototype translator from APPEL to an equivalent UMC specification, dubbed *Appel2UMC*, is written in OCaml, and structured in a syntax definition

module, a *Compiler*, and an *Unparser*. *Compiler* translates APPEL to UMC, at the abstract syntax level, and *Unparser* generates the textual version needed by the model checker. These core modules depend on a further one that defines the domain dependent features (triggers, conditions and actions), thus ensuring adaptability of the tool. At the moment, the syntax is about 100 lines, the core modules are slightly over 500 lines, and the domain dependent part less than 80 lines. Translation times are not an issue.

5.2 Preliminary Results

To evaluate the feasibility of the approach, we ran some experiments using the model checker UMC [23,19] to verify part of the policies introduced in [4] for structured parallel computations.

We consider two independently developed managers controlling respectively performance and power consumption of an application with a *farm* structure.

Managing Performance. The APPEL rules in Table 1 address *performance* management: the first two capture a noteworthy change in performance (trigger *NewPerformanceMonitored* and the others a noteworthy change in the parallelism degree of the execution. What a *noteworthy change* is, is defined by the semantics of the active sensors that generate the events matching these triggers.

These changes may be disregarded if they do not take the system outside of the "normal" operational range, i.e., when neither *LowPerformance* nor *HighPerformance* (*LowParDegree* nor *HighParDegree*, respectively) holds, that is, when the values returned by the corresponding sensors do not satisfy the intended condition.

Let us now consider what happens when PM1 fires, i.e., when performance drops below the threshold. The goal obviously being to reestablish an acceptable level, a new worker is introduced, in two macro steps, each sequencing two basic actions on the current state of the application. In the best of worlds, an available processor is allocated to the farm (*GetResource*), the appropriate run-time-support is deployed (*DeployRts*)and started (*StartRts*), and finally the new worker is linked (*LinkRts*)and therefore made available to the farm.

What if something goes wrong in the execute phase, e.g. no more processors are available? PM1 is written (like all the other rules in this simplistic scenario, by the way) using the composition operator **andthen** in such a way that the failure of any basic action entails the failure of the rule as a whole, and therefore the rule fires but has no effect whatsoever.[2]

The other rules were designed similarly, and use a few more basic actions, whose meaning should be immediate. Only *GetWorker* may need a comment: it selects one of the active workers in the farm, likely so that the manager can consequently free the resources it is using.

[2] Actually, care must be taken that the controlled application is rolled-back to its initial state. Also, in a realistic scenario, some alarm should be sent to the administrator, when appropriate.

Table 1. The Performance Manager Rules

PM1: **when** *NewPerformanceMonitored*
if *LowPerformance*
do (*GetResource* **andthen** *DeployRts*) **andthen**
(*StartRts* **andthen** *LinkWorker*)
PM2: **when** *NewPerformanceMonitored*
if *HighPerformance*
do (*GetWorker* **andthen** *UnlinkWorker*) **andthen**
(*StopRts* **andthen** *UndeployRts*)
PM3: **when** *NewParMonitored*
if *LowParDegree*
do (*GetResource* **andthen** *DeployRts*) **andthen**
(*StartRts* **andthen** *LinkWorker*)
PM4: **when** *NewParMonitored*
if *HighParDegree*
do (*GetWorker* **andthen** *UnlinkWorker*) **andthen**
(*StopRts* **andthen** *UndeployRts*)

Managing Power Consumption. The APPEL rules in Table 2 address this concern. They should be easily understandable, at this point, since they use many of the actions already used for performance management, but react to different events and are subject to new appropriate conditions.

The two rules deal only, in different ways, with the need to decrease power consumption. PCM1 takes a drastic approach, and kills one of the workers, to get the result. PCM2 attempts to save something, trading away one of the more power consuming workers for a less consuming one.

Conflict Definition. As we have seen, both managers operate on the application graph by executing actions like *LinkWorker* and *UnlinkWorker*, which include or remove a node in/from the current computation, respectively.

These actions are marked as an "atomic conflict", as they nullify each other, if performed in the same control cycle.

Putting UMC to Work. To illustrate how conflict detection is supported by UMC, we consider as simple a situation as possible, with only two rules, one from each manager, namely PM1 and PCM1. Given the parallel composition of these two rules, Appel2UMC generates the textual representation of the corresponding UMC model, dubbed *System* by default.

Loading *System* into the framework, an equivalent graphical representation (Fig. 4) is generated by the framework: it is a translation of the input *System* model into a standard automaton, resolving parallelism with interleaving: in this representation, "parallel" actions in the rules appear in sequence, in all possible different orders, along several traces. It is precisely the space of traces of this automaton that is searched by the UMC model checker.

Table 2. The Power Consumption Manager Rules

PCM1: **when** *NewPowerConsumptionMonitored*
if *PowerContractLow*
do (*GetWorker* **andthen** *UnlinkWorker*) **andthen**
(*StopRts* **andthen** *UndeployRts*)

PCM2: **when** *NewPowerConsumptionMonitored*
if *PowerContractLow*
do [(*GetPowerWorker* **andthen** *GetCheaperWorker*) **andthen**
(*UnlinkWorker* **andthen** *StopRts*)]
andthen
[(*UndeployRts* **andthen** *DeployRts*) **andthen**
(*StartRts* **andthen** *LinkWorker*)]

In this simple example it is clear, by inspection of the automaton, that the conflict will arise. However, as the number of rules in parallel increases, the size of the space of the traces of the corresponding automaton increases exponentially, and human inspection becomes quickly infeasible.

To use model checking instead, we need first to formalize the relevant question *may a conflict occur in one MAPE cycle?* in UCTL, in terms of traces: is there no trace among those generated by the automaton, which includes both *LinkWorker* and *UnlinkWorker*? Formally, the question is expressed by requiring that it should never be the case that there is a path were a *LinkWorker* (*UnlinkWorker*) state has a path to a subsequent *UnlinkWorker* (*LinkWorker*), that is:

$$\begin{array}{c}(\mathsf{not\ EF\ EX}\{\mathsf{LinkWorker}\}\ \mathsf{EF}\{\mathsf{UnlinkWorker}\}\ \mathsf{true})\\ \&\\ (\mathsf{not\ EF\ EX}\{\mathsf{UnlinkWorker}\}\ \mathsf{EF}\{\mathsf{LinkWorker}\}\ \mathsf{true})\end{array} \quad (1)$$

Our aim is to specify that there is no single MAPE cycle where both actions *LinkWorker* and *UnlinkWorker* are executed. The question has to be formulated in this way, since UMC translates the input model into a standard finite state machine, resolving parallelism with interleaving: "parallel" actions appear in sequence, in different orders, in several traces.

Running the model checker, it gives "false" as answer, and the explanation of this result gives the traces leading to the situation where the formula is demonstrated false.

To conclude we remark that the logical formulae associated with conflicts can be systematically written by the designer following the pattern of formula (1).

Resolving the Conflict. According to the method outlined in Sec. 5.1 we should be able to collect all the knowledge necessary to produce a modified

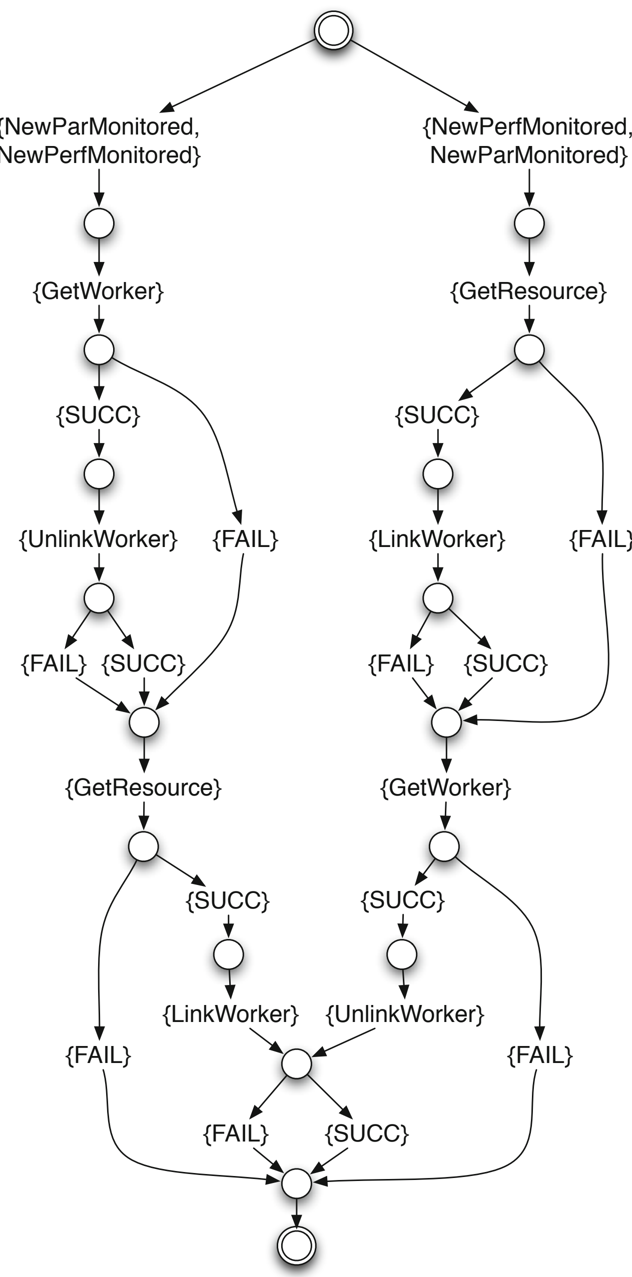


Fig. 4. Representation of *System* as an automaton

set of rules properly handling the conflict from the answer given by the model checker to explain why the model falsify formula (1):

- the situation leading to the conflict is determined by the contemporary firing of the power manager rule PCM1 to "reduce power usage", and of the performance manager rule PM1 to "increase parallelism degree".
- there is at least one path leading to the conflict, which includes the actions in PM1 and PCM1.

Based on this knowledge, we can conclude that handling of the detected conflict may be achieved by a high priority rule (or a set of rules):

- whose (new) trigger logically corresponds to the conjunction of the two triggers as APPEL, and most ECA based notation, does not support the conjunction of triggers but only trigger disjunction, and
- whose action part consists in a plan whose effect is an increase of the parallelism degree with reduced power consumption.

Alternatively, we may solve the conflict by assigning a priority to one of the conflicting rules, in such a way that only the highest priority rule is executed.

Feasibility. We discussed a very simple example: two rules giving rise to a very compact model and useful "explanations" in terms of traces. The number of states generated in the UMC model is below one hundred and the response time of the model checker is of the order of a fraction of a second.

We made a few slightly more realistic experiments using up to all the rules given above. The times needed to execute the model checker with different rules sets and queries are in the tens of milliseconds range: when 2, 4 or 6 rule systems are used, the time to model check the "conflict exists" formula are 30, 50 and 60 msecs, respectively (the model checker was running on a quad core Core Duo Intel Xeon workstation). When the AG(true) is model checked–this query gives the upper bound in execution times, as it requires the model checker to visit all possible paths in the model–the time spent in the model checker is 20, 120 and 250 milliseconds, respectively. These results seem to confirm that the approach is feasible in more realistic situations. We cannot show the involved automata, as the graphs are significantly larger and do not fit easily on a page.

6 Conclusions

We discussed formal tool support for the integration of independently developed autonomic managers, each taking care of a different non-functional concern. The formal tool support provides suitable hints to the programmer integrating these independently developed managers into a single parallel applications. As the manager programs are suitable sets of ECA rules, the formal tool support provides evidence of the conflicting rules in different managers as well as of the

initial situations (states) that eventually lead to the conflicting actions generated by the ECA rules. The preliminary results demonstrate the feasibility of the approach and the relatively modest computational cost of the model checker activities involved.

Paraphrase Perspective

The research results discussed in this paper will be exploited within the ParaPhrase project in various ways.

First, although not discussed here for the sake of simplicity, the ECA rule sets we are considering in our non-functional concern managers include rules that change the structure of the parallel computation. For example, a manager taking care of performance in a program whose parallel structure may be represented as a *pipeline(seq(f), seq(g), seq(h))* may discover that the second stage takes much longer to execute than the first and the third. Therefore he may execute an action aimed at transforming the program into a *pipeline(seq(f), farm(seq(g)), seq(h))*. As the main focus of ParaPhrase is on parallel program refactoring, these rules transforming parallel pattern compositions to better performing parallel pattern compositions represent natural candidates for use in the refactoring process.

Second, we foresee the possibility to implement some kind of dynamic management of the re-factoring to suit the varying conditions on the target architectures within ParaPhrase. The techniques discussed here will naturally suit the need to verify that no conflicts are generated while dynamically re-factoring our parallel applications.

References

1. Aldinucci, M., André, F., Buisson, J., Campa, S., Coppola, M., Danelutto, M., Zoccolo, C.: Parallel program/component adaptivity management. In: Gorlatch, S., Danelutto, M. (eds.) Proc. of the Integrated Research in Grid Computing Workshop, Pisa, Italy, TR-05-22, pp. 95–104. Università di Pisa, Dipartimento di Informatica (2005)
2. Aldinucci, M., Campa, S., Danelutto, M., Dazzi, P., Kilpatrick, P., Laforenza, D., Tonellotto, N.: Behavioural skeletons for component autonomic management on grids. In: CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments, Heraklion, Crete, Greece (June 2007)
3. Aldinucci, M., Danelutto, M., Kilpatrick, P.: Autonomic management of non-functional concerns in distributed and parallel application programming. In: Proc. of Intl. Parallel & Distributed Processing Symposium, IPDPS, Rome, Italy, pp. 1–12. IEEE (May 2009)
4. Aldinucci, M., Danelutto, M., Kilpatrick, P.: Autonomic managenemt of multiple non-functional concerns in behavioural skeletons. In: Proc. of the CoreGRID Symposium 2009, CoreGRID, Delft, The Netherlands. Springer (August 2009)
5. Alloy Community, `http://alloy.mit.edu/community/`
6. ter Beek, M., Gnesi, S., Montangero, C., Semini, L.: Detecting policy conflicts by model checking UML state machines. In: Reiff-Marganiec, S., Nakamura, M. (eds.) Feature Interactions in Software and Communication System X, pp. 59–74. IOS Press (2009)

7. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: An Action/State-Based Model-Checking Approach for the Analysis of Communication Protocols for Service-Oriented Applications. In: Leue, S., Merino, P. (eds.) FMICS 2007. LNCS, vol. 4916, pp. 133–148. Springer, Heidelberg (2008)
8. Calder, M., Kolberg, M., Magill, E.H., Marples, D., Reiff-Marganiec, S.: Hybrid solutions to the feature interaction problem. In: Amyot, D., Logrippo, L. (eds.) FIW, pp. 295–312. IOS Press (2003)
9. Calder, M., Magill, E.H.: Feature Interactions in Telecommunications and Software Systems VI, Glasgow, Scotland, UK, May 17-19. IOS Press (2000)
10. Danelutto, M., Kilpatrick, P., Montangero, C., Semini, L.: Model Checking Support for Conflict Resolution in Multiple Non-functional Concern Management. In: Alexander, M., D'Ambra, P., Belloum, A., Bosilca, G., Cannataro, M., Danelutto, M., Di Martino, B., Gerndt, M., Jeannot, E., Namyst, R., Roman, J., Scott, S.L., Traff, J.L., Vallée, G., Weidendorfer, J. (eds.) Euro-Par 2011 Workshops, Part I. LNCS, vol. 7155, pp. 128–138. Springer, Heidelberg (2012)
11. Layouni, A.F., Logrippo, L., Turner, K.J.: Conflict detection in call control using first-order logic model checking. In: du Bousquet, L., Richier, J.-L. (eds.) Proc. 9th Int. Conf. on Feature Interactions in Software and Communications Systems, France, pp. 77–92. IMAG Laboratory, University of Grenoble (2007)
12. Mazzanti, F.: UMC User Guide v3.3. Technical Report 2006-TR-33, Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", CNR (2006)
13. Montangero, C., Reiff-Marganiec, S., Semini, L.: Logic–Based Detection of Conflicts in Appel Policies. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 257–271. Springer, Heidelberg (2007)
14. Montangero, C., Reiff-Marganiec, S., Semini, L.: Logic-based conflict detection for distributed policies. Fundamenta Informaticae 89(4), 511–538 (2008)
15. Montangero, C., Semini, L.: Distributed states logic. In: 9th International Symposium on Temporal Representation and Reasoning, TIME 2002, Manchester, UK. IEEE CS Press (July 2002)
16. Montangero, C., Semini, L., Semprini, S.: Logic Based Coordination for Event–Driven Self–Healing Distributed Systems. In: De Nicola, R., Ferrari, G.-L., Meredith, G. (eds.) COORDINATION 2004. LNCS, vol. 2949, pp. 248–262. Springer, Heidelberg (2004)
17. Reiff-Marganiec, S., Turner, K.J.: Feature interaction in policies. Comput. Networks 45(5), 569–584 (2004)
18. Ruz, C.: Autonomic Monitoring and Management of Component-Based Services, PhD Thesis. Univ. de Nice - Sophia Antipolis (2011)
19. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: A state/event-based model-checking approach for the analysis of abstract system properties. Sci. Comput. Program. 76(2), 119–135 (2011)
20. Tonti, G., Bradshaw, J.M., Jeffers, R., Montanari, R., Suri, N., Uszok, A.: Semantic Web Languages for Policy Representation and Reasoning: A Comparison of KAoS, Rei, and Ponder. In: Fensel, D., Sycara, K., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 419–437. Springer, Heidelberg (2003)
21. Turner, K.J., Reiff-Marganiec, S., Blair, L., Pang, J., Gray, T., Perry, P., Ireland, J.: Policy support for call control. Computer Standards and Interfaces 28(6), 635–649 (2006)
22. Turner, K.J., Reiff-Marganiec, S., Blair, L., Cambpell, G.A., Wang, F.: Appel: An adaptable and programmable policy environment and language. Technical Report TR-161, University of Stirling (December 2009)
23. UMC v3.5, http://fmt.isti.cnr.it/umc

The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems

Kevin Hammond[1], Marco Aldinucci[2], Christopher Brown[1], Francesco Cesarini[3], Marco Danelutto[4], Horacio González-Vélez[5], Peter Kilpatrick[6], Rainer Keller[7], Michael Rossbory[8], and Gilad Shainer[9]

[1] School of Computer Science, University of St Andrews, Scotland, UK
[2] Computer Science Dept., University of Torino, Torino, Italy
[3] Erlang Solutions Ltd., London, UK
[4] Dept. Computer Science, Universitá di Pisa, Pisa, Italy
[5] School of Computing, Robert Gordon University, UK
[6] School of Electronics, Electrical Eng. and Comp. Sci., Queen's Univ. Belfast, UK
[7] High Performance Computing Centre, Stuttgart (HLRS), Germany
[8] Software Competence Centre Hagenberg, Austria
[9] Senior Director of HPC and Technical Computing, Mellanox Technologies, Israel
{kh,chrisb}@cs.st-andrews.ac.uk, aldinuc@di.unito.it,
francesco@erlang-solutions.com, marcod@di.unipi.it,
h.gonzalez-velez@rgu.ac.uk, p.kilpatrick@qub.ac.uk, keller@hlrs.de,
michael.rossbory@scch.at, Shainer@Mellanox.com

Abstract. This paper describes the ParaPhrase project, a new 3-year targeted research project funded under **EU Framework 7 Objective 3.4 (Computer Systems)**, starting in October 2011. ParaPhrase aims to follow a new approach to introducing parallelism using advanced refactoring techniques coupled with high-level parallel design patterns. The refactoring approach will use these design patterns to restructure programs defined as networks of software components into other forms that are more suited to parallel execution. The programmer will be aided by high-level cost information that will be integrated into the refactoring tools. The implementation of these patterns will then use a well-understood algorithmic skeleton approach to achieve good parallelism.

A key ParaPhrase design goal is that parallel components are intended to match *heterogeneous* architectures, defined in terms of CPU/GPU combinations, for example. In order to achieve this, the ParaPhrase approach will map components at link time to the available hardware, and will then re-map them during program execution, taking account of multiple applications, changes in hardware resource availability, the desire to reduce communication costs etc. In this way, we aim to develop a new approach to programming that will be able to produce software that can adapt to dynamic changes in the system environment. Moreover, by using a strong component basis for parallelism, we can achieve potentially significant gains in terms of reducing sharing at a high level of abstraction, and so in reducing or even eliminating the costs that are usually associated with cache management, locking, and synchronisation.

B. Beckert et al. (Eds.): FMCO 2011, LNCS 7542, pp. 218–236, 2013.
© Springer-Verlag Berlin Heidelberg 2013

1 Introduction

From the 1960s until very recently, hardware designers were able to exploit the effects of Moore's law to create processors with ever-increasing clock frequencies. Software benefited from each new processor generation more or less automatically. At the same time, software engineers remained essentially wedded to the inherently sequential *von Neumann* programming model that has been in use ever since the early days of computing. Most of the major advances in programming language technology and software engineering that have taken place (e.g. structured programming, object-orientation, or abstract modelling) were therefore solely motivated mainly by the need to keep ever-larger software systems manageable, rather than to make effective use of the available hardware capabilities. This situation is currently changing, however, *and changing extremely rapidly.* Future multicore/manycore hardware will not be *slightly parallel*, like today's dual-core and quad-core processor architectures, but will be *massively parallel.* Concurrently with this trend towards increasing numbers of cores, there is also a strong trend towards *heterogeneous* architectures, with chips containing not only conventional processor cores, but also various specialist processing units such as graphics-processing units (GPUs), physics engines, digital signal processors (DSPs), etc. Properly exploiting heterogeneous multicore technology is essential for today's users of high-performance computers: provided they can be properly harnessed, hybrid multicore/manycore systems offer the potential for cheap, scalable and energy-efficient high-performance computing. Unfortunately, while GPU computing [52] compares very favourably with multicore CPUs in terms of performance, *it has even worse programmability.* It is therefore becoming increasingly obvious that the traditional sequential programming model has reached its limits. This problem of programmability for future parallel computers motivates the ParaPhrase project.

The Challenge. It is clear that effectively exploiting *heterogeneous* multicore/manycore processor technology will be an essential requirement for future software developers. The main challenge they face is finding a programming model that provides a suitable level of abstraction, while still allowing good use of the available hardware resources. It is already very difficult for classically-trained applications programmers to benefit from the performance offered by today's multicore systems, and only highly-skilled programmers or those seeking the highest levels of performance are presently exposed to parallel programming techniques [1]. Without a fundamental shift in the programming model, programmers will find it essentially impossible to exploit the mid-term/long-term developments that major hardware companies such as Intel and NVidia promise to deliver. The dilemma is that a large percentage of mission-critical enterprise applications will not "automagically" run faster on multicore servers. In fact, many will actually run slower [43]. It is therefore essential that we make it as easy as possible for applications programmers to exploit the latest developments in heterogeneous multicore/manycore architectures, while still making it easy to target future (and perhaps unanticipated) hardware developments.

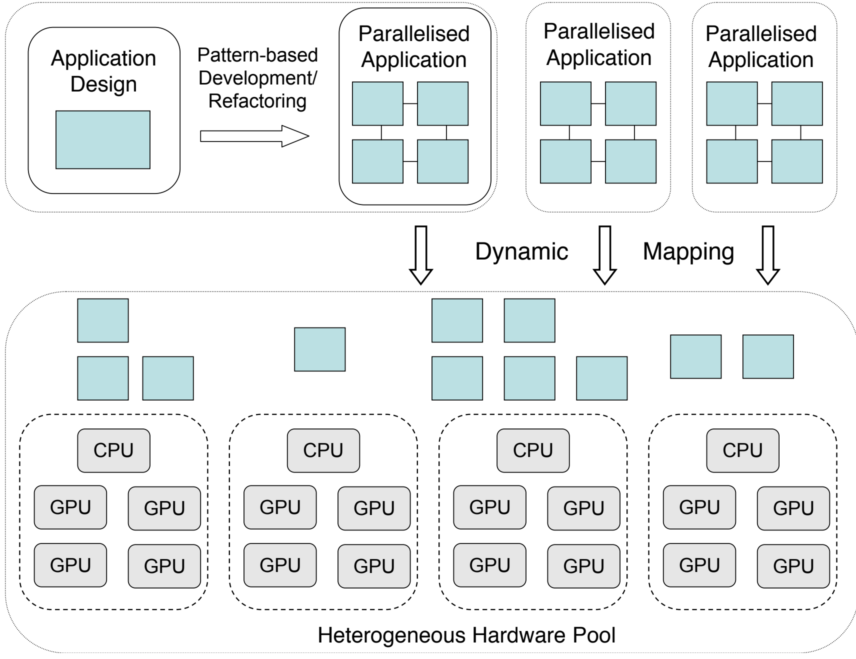


Fig. 1. The ParaPhrase Vision

2 Related Work

Pattern-Based Parallel Programming Models. The recent advent of multi-core processors, GPUs, chip multiprocessors, and multinode clusters and constellations has dramatically increased the number of concurrent processors that are available within a single system configuration. Architectures involving dozens of heterogeneous core in an integrated processing node are becoming commonplace in high-performance computing environments, and, as a result, state-of-the-art supercomputing facilities must efficiently administrate thousands of processing units. JUGENE (the Blue Gene/P PRACE platform at Julich) features 294,912 "traditional" multicore processing elements and the Tianhe-1 comprises 186,368 CPU/GPUs, according to the November 2010 Top500 list [60]. However, software development techniques have not evolved at the same pace and the software itself has typically outlived architectural generations. Linpack, the canonical HPC benchmark, was initially released in 1979 and still constitutes the basis for the Top500 list. While the *linguae franca* for large parallel computers have historically been Fortran and C coupled with coordination libraries such as MPI, OpenMP, or PVM, there is no clear trend on how to efficiently engineer the required parallel programs for mainstream parallel computers. Research effort needs to be devoted to design efficient parallel programs that not only exploit the architectural characteristics of parallel architectures, but also preserve the

investment in programming over time in a number of platforms. The challenge is therefore to holistically develop high-quality parallel software, which is not only efficient and scalable, but which is also well-programmed and generic.

While they were originally defined as abstractions of common themes in object-oriented programming [25,26], design patterns have subsequently been incorporated into parallel programming methodologies [50]. Pattern-based parallel programming allows an application programmer the freedom to generate new parallel programs by parameterising parallel abstractions. This approach is very similar to that taken by algorithmic skeletons, described below. The main differences lie in their respective visions [18]: in the design pattern approach the (software engineering driven) vision proposes patterns as *models* to be instantiated, implemented and used by the programmer; in the algorithmic skeleton approach the (language-driven) vision uses skeletons as predefined *language constructs* or library entries that can be seamlessly used by the programmer in the same way as other non-parallel language constructs and/or library entries.

The parallel programming pattern concept has been extended into a design method under the umbrella of *parallel pattern languages*. Unlike other parallel programming languages, parallel pattern languages present rules for designing parallel programs based on problem-class abstractions which describe parallel structure, dataflow, and communication; critical region locks, such as test-and-set and queued for simple mutual exclusion, or reader/writer for concurrent execution; or socket-based operators for web applications.

In their frequently-cited report, Asanovic et al. have emphatically suggested the deployment of parallel design patterns to successfully produce effective parallel programs for multi and manycore architectures [7]. However, the generic implementation of these parallel design patterns in multi and manycore architectures requires the use of refined techniques which can provide a clear-cut separation of software and hardware. This has long been considered critical to the success of any parallel programming endeavour since it is essential if we are to foster the reuse of algorithms and software. Moreover, we consider that the division of the structure from the application itself is crucial to the goal of delivering adaptability.

Algorithmic Skeletons. *Algorithmic skeletons* abstract commonly-used patterns of parallel computation, communication, and interaction into a set of language constructs [17,32]. Skeletons present a top-down structured approach where parallel programs are formed from the parametrisation of skeleton nest, also known as structured parallelism. Structured parallelism deployed through skeleton frameworks provides a clear and consistent structure across platforms by distinctly decoupling the application from the structure in a parallel program. It does not rely on any specific hardware and it benefits entirely from any performance improvements in the system infrastructure. Algorithmic skeleton frameworks (provided either as new languages or as libraries) have been implemented using a variety of techniques including macro data flow, templates, aspect-oriented programming, and rewriting techniques to target distributed

architectures (such as COW/NOWs and grids) and, more recently, homogeneous multicore architectures. Recently, implementation techniques supporting *expandable* algorithmic skeleton sets have been demonstrated [3]. It is arguable that the different techniques used to implement algorithmic skeleton frameworks are suitable to support implementations targeting heterogeneous multicore architectures. While algorithmic skeletons (and indeed parallel patterns in general) cannot be used to produce *all* parallel and distributed programs, there is a growing number of important applications [19,55]. A number of recent and highly successful "programming models" such as the well-known *Google MapReduce* also derive and inherit from algorithmic skeletons [13]. Furthermore, skeletal methodologies inherently possess a predictable communication and computation structure, since they directly capture the structure of the program. They therefore provide, by construction, a foundation for performance modelling and estimation of parallel applications.

Refactoring Technology for Parallel Programming. Refactoring [51] changes a program's structure, but keeps its behaviour the same. *Software refactoring* involves using tool support to adapt or change existing software according to well-defined patterns. It is primarily used to produce code that is either more efficient, that uses specific library/language capabilities, or that is better structured to meet some software engineering goals. The primary challenges lie in: i) identifying the refactorings that are available to the programmer; ii) guiding the programmer in determining which of those refactorings are most sensible/beneficial; and iii) correctly applying the refactoring so that the resulting code has the required behaviour without introducing unwanted changes in functionality. Refactoring tools such as Eclipse [24] now offer an extensive range of refactorings also including inlining, extract constant, introduce parameter and encapsulate a field. Many refactoring tools are fully-fledged commercial or open-source products.

Refactoring Parallel Programs. Despite the obvious advantages, there has so far been little work in the field of applying software refactoring technology to assist parallel programming. The earliest work on interactive tools for parallelisation stemmed from the Fortran community, targeting loop parallelisation [40]. These interactive tools were early transformation engines allowing users to manipulate loops in their Fortran programs by specifying what loops to interchange, align, replicate or expand. The interactive tools typically reported to the programmer various information such as dependance graphs, and was mainly applied to the field of numerical computation. Recent work in the field includes Reentrancer [62]: a refactoring tool developed by IBM for making code reentrant. Reentrancer targets global data by making them thread-safe. Further recent work includes a refactoring approach to parallelism by Dig [20], targeted at introducing concurrency in Java programs by aiming to make them more thread safe, increasing throughput and scalability. Hitherto, Dig's refactoring tool contains a minor selection of transformations including *make class immutable*, *parallelise loop* and *convert HashMap to ConcurrentHashMap*. Software refactoring

techniques have therefore only previously been applied in a very limited parallel setting: by applying simple transformations to introduce parallel loops and thread safety in object-oriented (OO) programs. Currently, these approaches do not take any extra function properties into account, such as hardware characteristics, costing and profiling, for aiding the refactoring process. Furthermore, the techniques are rather limited to homogeneous architectures and OO languages, rather than applying general patterns to heterogeneous architectures, as needed in the ParaPhrase project.

Automatic Parallelisation. (Semi-)Automatic parallelisation poses two main challenges: i) how to identify those parts of the program that could be executed concurrently; and ii) how to map these parts onto a given set of computing resources. Failing in either challenge immediately limits any potential performance gains. Both of these challenges are clearly addressed within the ParaPhrase project. Most research on automatic parallelisation has focused on how to identify concurrency. Many sophisticated optimisation techniques based on dependence analyses have been developed [8,63,6]. They form the basis for the *polytope model* [42,23,9], which facilitates compiler-directed transformations to increase loop-level concurrency. Such approaches are fundamentally limited, however, to specific programming patterns, and tend to favour fine-grained parallelism. In the ParaPhrase project, we take a higher-level approach to identifying parallelism, recognising that programmer assistance and insight may be valuable at this stage. This has three main advantages: firstly, the pattern-based approach allows us to easily decompose the application into suitably concurrent tasks, *breaking accidental dependencies that will limit the polytope approach*; secondly, we can use performance information not only to drive the choice of parallel implementation, as commonly happens, but also to guide programmer-directed refactoring to identify the most profitable parallel structure; and thirdly, by using a component structure with a strong explicit resource interface that automatically exposes necessary inter-task dependencies, and avoids accidental dependencies that restrict the opportunities for concurrency. Having identified good parallelism, it is necessary to focus on the issues involved in the second challenge, i.e. *effectively* mapping concurrency onto the available computing resources so as to maximise performance. This mapping requires decisions to be made such as: Which concurrently executable parts should actually be done in parallel? How and when should synchronisation happen? Where should data be placed? What layout in memory should be used for the data to enable non-conflicting (and cache-friendly) concurrent access? etc.

A large body of work addresses these issues in the context of nested loops. Besides scheduling-driven [22,38] and partitioning-driven [44,45] approaches, more recent tiling-based [12] and streamisation-based [53] approaches have shown promising results for shared-memory architectures. Any compiler-driven decision mechanism for these aspects relies on the availability of as precise as possible knowledge of application properties such as typical data sizes, function application frequencies, typical parameter ranges etc. Over recent years we have

developed several analysis and program transformation techniques that aim at identifying these properties. Amongst these are partial evaluation techniques such as those described in [36,59,10,39], as well as code restructuring techniques [56,34,37], and multi-threaded code generation techniques [33,35]. However, the interplay of such optimisations combined with the complexity and variety of target platforms often renders static mapping approaches far less effective than the mappings that can easily be achieved manually. Even in the single-processor setting it has been shown that semi-static approaches such as iterative optimisations [47,54] can improve the effectiveness of the optimisation process. The ParaPhrase approach avoids these problems by exploiting a more dynamic approach that can adapt to changing system conditions, given a good initial placement as its starting point.

Hardware/Software Virtualisation. In the context of ParaPhrase, the purpose of the hardware/software virtualisation layers is: i) to abstract over the available heterogeneous multicore hardware in order to support the automated mapping of an application onto diverse targets; ii) to support dynamic remapping and adaptivity; and iii) to support the seamless mapping of multiple simultaneous parallel applications to the available hardware resources. This represents a significant challenge. The virtualisation must allow the decomposition of the parallel software into units that can easily be mapped/re-mapped to alternative hardware realisations; it must support cost information that allows rapid decisions to made on dynamic re-mapping; it must be sufficiently flexible that it can support all the required parallel patterns; and it must be sufficiently lightweight that it does not impose excessive overhead that may restrict the flexibility of dynamic re-mapping.

The state-of-the-art in dynamic targeting is epitomised by the Java Virtual Machine (JVM) [46], where compiled code, represented as an abstract instruction set (Java byte-code), is either interpreted by the JVM or is compiled on execution into the instruction set of that processor. Virtualisation can also be used to translate between instruction sets. For example, the full virtualisation of a target is possible where a virtual machine environment is able to execute all software that is capable of executing on that target. A good example is the Transmeta Crusoe architecture [27], which provides a full virtualisation of the x86 platform onto a much more energy-efficient VLIW processor. The techniques exploited here use a mixture of binary translation from native x86 binaries to the Crusoe's VLIW instruction set together with a mechanism for caching recently translated code blocks. These techniques, namely interpretation, binary translation and just-in-time compilation may be augmented with the use of fat binaries, where the choice of target or target parameterisation is reasonably bounded. All of these techniques could be exploited by ParaPhrase. However, the main issue for ParaPhrase is the efficient execution of generic parallel code on an arbitrary heterogeneous target. Although the Java execution model supports concurrency, this model was originally designed to support threaded programs on a single processor rather than supporting distributed parallel programming. Moreover,

the model is not constrained, which means that the programmer must ensure both that threads do not interfere with each other and that resources are properly synchronised. It therefore does not meet the objectives posed by ParaPhrase. It follows that a more general model of concurrency is required, one that ideally is safely composable without inducing deadlock or compromising efficiency. By using a new virtualisation model based around costable software components coupled with a simple hardware virtualisation, we anticipate that we will be able to meet the stringent requirements of the ParaPhrase project.

Autonomic and Dynamic Placement. Placement of concurrent components derived from the compilation of high level parallel patterns on multicore, heterogenous architectures poses different problems related to efficiency and performance. Vadhiyar and Dongarra [61] suggest that a "self-adaptive software system examines the characteristics of the computing environments and chooses the software parameters needed to achieve high efficiency on that environment". Thus, we consider that the key challenges in adaptively improving the performance of parallel programs in a heterogeneous system are therefore:

1. the correct selection of resources (processors, links) from those available;
2. the correct adjustment of algorithmic parameters (for example, blocking of communications, granularity); and, most importantly,
3. the ability to adjust all of these factors *dynamically* in the light of evolving external pressure on the chosen resources.

Although different parallel solutions for heterogeneous distributed systems have traditionally exhibited parallel patterns, their associated optimisations have not necessarily exploited the application structure. They have either modified the scheduler [14] or kept the actual application interlaced [57], without decoupling the structure from the behaviour. Such challenges are aligned with the traditional view on intra-application scheduling, which proposes five actions: i) select resources to schedule the tasks; ii) map tasks to processors; iii) distribute data; iv) order tasks on compute resources; and, v) order communication tasks. However, traditional strategies for placement or scheduling in distributed systems [11,15,21,41,49] rely on system simulators, dedicated configurations, and/or performance estimators to model the general system, particularly to characterise the background load in terms of its job arrival rate. While much can be said about the reproducibility of their results, one may argue that they artificially create tractable evaluation scenarios for their scheduling policies. It follows that he ParaPhrase methodology cannot be simplistically compared to any task scheduling policy in terms of algorithmic optimality and complexity, but ought to be evaluated in terms of the makespan for a certain workload.

In addition to the preliminary, possibly static, optimisations performed when deploying the program components onto the target architecture, based on the expected performance models of the parallel patterns used, complementary approaches will be considered:

Control Loops. This approach extends the experience of the Universities of Pisa and Torino in Behavioural Skeletons [5,2,4].

Divisible Workloads. This approach builds on previous work from several partners including that of Robert Gordon University on statistical scheduling of divisible workloads [30] and the systematic introduction of adaptivity into parallel patterns and skeletons [29,31,4].

Each of the components derived from the compilation of the high-level patterns will be equipped with a couple of additional interfaces: a *sensor* interface and an *actuator* interface. The former will provide methods suitable to gather actual measures related to the current status of the computation (e.g. throughput, service time, latency). The latter will provide methods to *modify* the implementation of the high level pattern (e.g. change its parallelism degree by adding/removing components, migrating the component from CPUs to GPUs (or *vice-versa*). An additional *autonomic performance manager* component will be added to the implementation of each high level parallel pattern. The manager will implement a control loop. Performance of the parallel pattern implementation will be monitored through the sensor interface and possibly actions performed by invoking the actuator methods to improve overall pattern performance. The autonomic manager may be implemented on top of a business rule engine in such a way the rules executed at each control loop iteration may embody all techniques to dynamically optimise the performance of the high level patterns as well as any new and/or experimental techniques. In particular, techniques based on learning from previous experience may be implemented, provided a data base of past computation management is maintained.

In summary, the ParaPhrase approach can be categorised as a autonomic and dynamic placement methodology for parallel programs executing in heterogeneous distributed systems, which is:

dynamic since the correct selection of resources and the adjustment of algorithmic parameters are performed at execution time;

autonomic due to the provision of intrinsic mechanisms to dynamically adjust to variations in system performance;

application-level because all decisions are based on the specific requirements of the application at hand; and,

heuristic because it comprises a set of rules intended to increase the probability of enhancing the overall parallel program performance.

3 The ParaPhrase Project

The challenges identified in Section 1 require a new and radical approach that tackles parallel programming in a coherent and holistic way. The ParaPhrase project aims to produce a new structured design and implementation process for heterogeneous parallel architectures, where developers exploit a variety of parallel patterns to develop component-based applications that can be mapped to the available hardware resources, and which may then be dynamically re-mapped to

meet application needs and hardware availability (Figure 1). We will exploit new developments in the implementation of parallel patterns that will allow us to express a variety of parallel algorithms as compositions of lightweight software components forming a collection of virtual parallel tasks. Components from multiple applications will be instantiated and dynamically allocated to the available hardware resources through a simple and efficient software virtualisation layer. In this way, we will promote adaptivity, *not only at an application level, but also at a system level.* Finally, virtualisation abstractions will be provided across the hardware boundaries, allowing components to be dynamically re-mapped to either CPU or GPU resources on the basis of suitability and availability.

3.1 Achieving the ParaPhrase Project Vision

In order to achieve the vision described above so that we can make effective use of recent and future advances in heterogeneous multicore/manycore computing, a number of key technical problems must be addressed.

We Must Develop New Parallel Programming Models. The parallel programming models in widespread use today require the programmer to manage many low-level organisational details, including communication, placement and synchronisation. This low-level coding style makes programming parallel systems notoriously difficult, since a whole new class of programming errors severely impacts programming productivity as mismatched communications, deadlocks, race conditions, which often exhibit non-deterministic behaviour

We Must Develop New Means of Identifying Parallelism. Low-level parallelism libraries, such as OpenMP, MPI or Pthreads are widely used in non-numerical applications. However, such approaches are highly inflexible, making it hard: to dynamically adapt to the execution environment; to introduce the high-level changes to program structure that may be necessary to support new multicore computer architectures; or to refactor existing program code to support a new parallel application. What is needed is a simpler way of identifying parallelism that can both support adaptation to conform to dynamic changes in hardware availability and that can support long-term software evolution to meet new application or hardware needs.

We Must Develop New Ways of Abstracting across the Capabilities of Different Architectures/Devices. Code written for a general-purpose GPU (GPGPU) cannot easily be executed on a general-purpose CPU core, or *vice-versa.* This limits the ability to reconfigure software to exploit the available hardware resources, effectively restricting software to a static placement.

The ParaPhrase project focuses on issues of programmability for parallel systems. It will develop a new approach based on patterns of parallelism that structure independent parallel components. Each independent component will have a well-defined interface identifying memory dependencies and key extra-functional properties, such as performance information, degree of parallelism and communication access patterns. The patterns and components will be identified using

a high-level and novel *refactoring* approach that will guide the programmer towards optimal design decisions targeting a range of different implementations. Having chosen a pattern, the pattern must be *mapped* to the available hardware targets using a *behavioural skeleton* approach. Finally, since it is well known that even an optimal initial placement is unlikely to achieve maximum performance under real-world operating conditions, components from multiple applications will be *re-mapped* during execution in order to maximise performance. The project thus combines automatic approaches to task placement and re-mapping with an assisted approach to the initial identification of the parallel program structure. The use of a component-based approach, with strong behavioural interfaces forming a virtual parallel task structure, is fundamental in allowing the ParaPhrase project to achieve its goals.

3.2 Key Technologies

The key technologies used in and deployed by the ParaPhrase project are:

Refactoring: the process of changing the structure of a program without changing its behaviour. Refactoring has been practised implicitly for as long as programs have been written, as programmers actively re-structure their code as they typically build software. Refactoring tools, such as the Paraphrase Refactoring Tool, will provide a set of well-defined semi-automatic refactorings aimed at parallelisation that will allow the programmer to simply select which parallel pattern (or skeleton) to apply. The refactoring tool then checks any conditions and applies the transformations automatically.

Virtualisation: the ParaPhrase project deploys two levels of virtualisation. *Component Virtualisation* abstracts over different software implementations of the same parallel program, allowing parallel programs to be composed from several software components. *Hardware Virtualisation* abstracts over the heterogeneous hardware resources that are available, allowing software components to be mapped/re-mapped to alternative physical hardware.

Parallel Patterns: high-level expressions of generalised parallel algorithms that typify *classes* of parallel problems. Typical parallel patterns include task farms, work pools, pipelines, parallel maps and parallel reduce operations.

Skeletons: "implementations" of design patterns. A skeleton provides a parametric implementation of a specific parallel design pattern targeting a given class of architecture.

FastFlow: a skeleton-based programming framework, developed at the Universities of Pisa and Torino, that efficiently targets cache-coherent multicore architectures.

Erlang: a strict functional language with dynamic types and support for built-in concurrency. Erlang is executed on the Erlang Virtual Machine layer, which provides implicit mechanisms for managing fault tolerance and for deploying data serialisation in message passing.

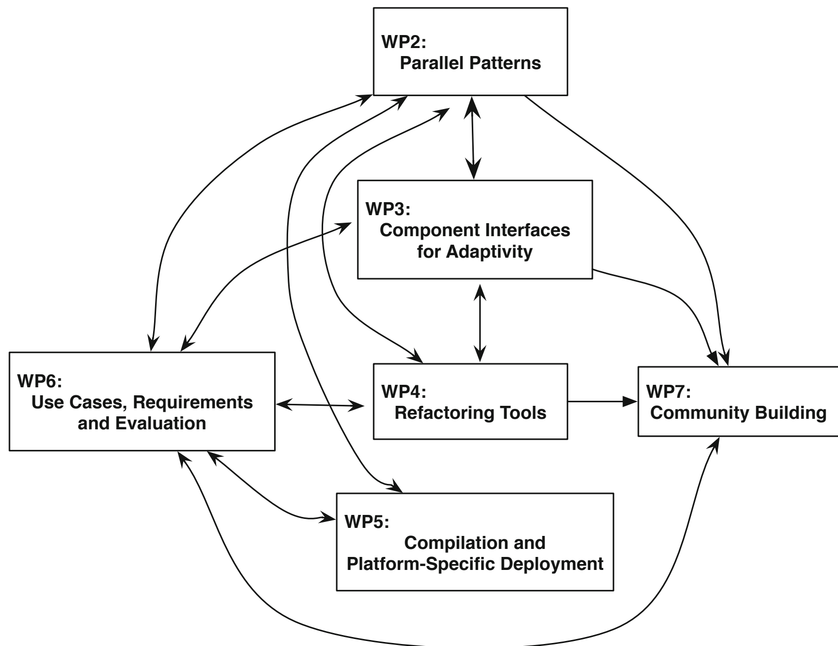


Fig. 2. The ParaPhrase Workpackages and their Dependencies

3.3 ParaPhrase Structure and Workplan

An outline structure of the ParaPhrase project into its component technical workpackages (WP2–WP7; WP1 is Management) is shown in Figure 2. WP2 covers high-level parallel patterns and their implementation as skeletons. WP3 defines the software virtualisation framework. WP4 develops new refactoring tools and techniques. WP5 considers adaptive mapping technology. WP6 validates the work done in the project against some real applications. Finally, WP7 aims to develop a user community for the ParaPhrase technologies.

High-Level Parallel Patterns (WP2). The use of a pattern-based approach allows parallelism to be expressed at a very high level of abstraction, so achieving the overall aim of simplifying the programming of multicore systems. At the same time, by exposing parallelism in terms of specific parallel patterns, parallel programs can be easily refactored into alternative forms with different parallel behaviours, as indicated in Figure 3. For example, a parallel *map* operation, where a single operation is applied in parallel to every element of a collection of data may be either refactored into a parallel *task farm*, with a fixed mapping of tasks to processing elements; or may alternatively be refactored to a parallel *workpool*, where the tasks are mapped dynamically to processing elements as

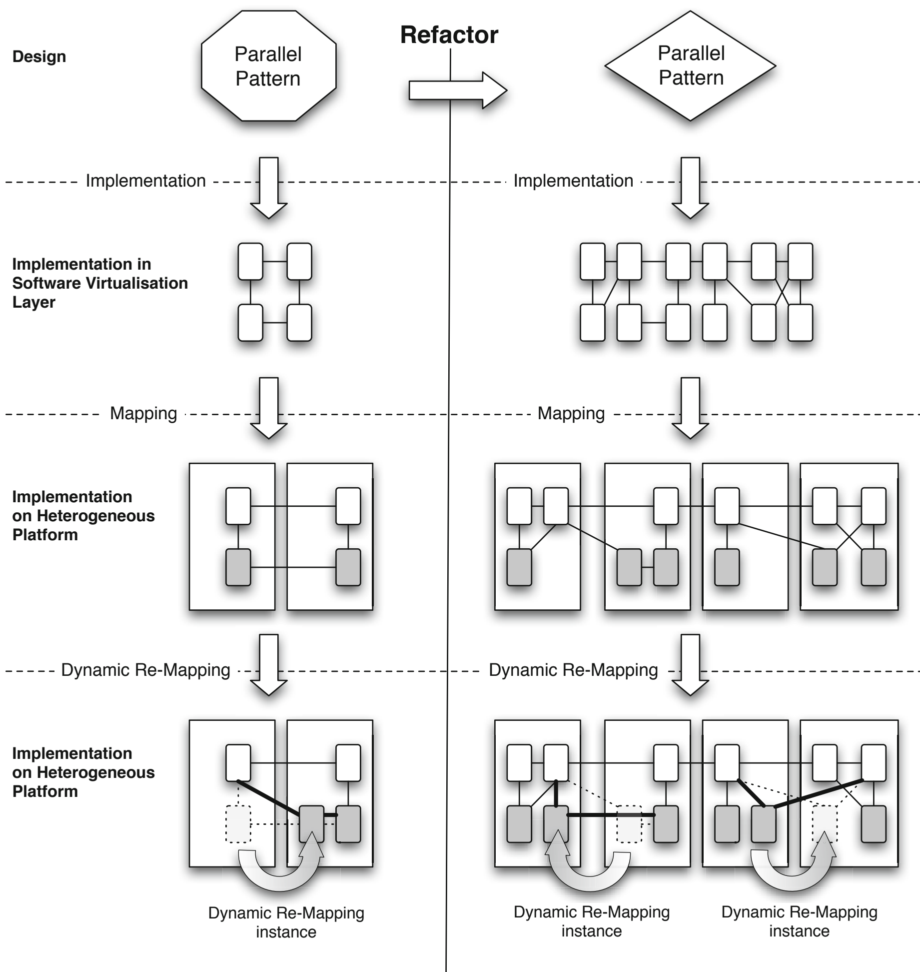


Fig. 3. The ParaPhrase Approach: Refactoring and Implementing Parallel Patterns

they become available. Depending on the structure of the parallel application, one or other of these patterns may be preferable: for example, a task farm is more suitable for more regular task sizes; where a workpool is more suitable for irregular task sizes. Achieving this objective will therefore allow us to improve programmability of multicore systems.

Heterogeneous Pattern Implementations (WP2). Heterogeneity and parallelism are critical to future high-performance computers: future computer architectures are likely to be built around collections of large numbers of parallel processing elements, comprising both general-purpose units (CPUs), but also higher-performance but more specialised units (e.g. GPUs, DSPs, Physics

Engines, FPGAs, ASICs etc). These units may have overlapping, but not interchangeable capabilities. These units may be grouped into different configurations, comprising different ratios of general-purpose to special-purpose units, different clock speeds etc. In order to make effective use of the available processing elements in such an architecture, it is therefore essential to consider heterogeneity. Achieving this objective will thus allow us to demonstrate the benefits of using high-level patterns for heterogeneous multicore systems in future high-performance computing applications.

Software Virtualisation Framework (WP3). Once a parallel program has been refactored into the required parallel pattern, it can then be decomposed into a set of cooperating parallel components, with well-defined communication interfaces, and interconnections using an *algorithmic skeleton* approach (Section 2). This componentisation and encapsulation is important, since by providing alternative implementations of a component, that component can be mapped to different kinds of hardware processing elements, for example to either a CPU or a GPU. Since the use of high-level patterns will allow programs to be decomposed into potentially large numbers of parallel components, and we will be able to use the same hardware to execute components from multiple parallel applications, we will have a great deal of flexibility both when initially mapping components to CPU/GPU elements, and in subsequently re-mapping components to CPU/GPU cores as a result of dynamic changes in the execution environment.

Refactoring Tools for Parallel Patterns (WP4). Refactoring tools support programmer-directed transformation of the source code in order to improve behavioural or other properties of program code. In the ParaPhrase project, we are interested in *supporting* the programmer by allowing them to choose between alternative parallel patterns, using high-level information about their run-time behaviours. While it would be, in principle, possible to automatically map high-level programming patterns directly to implementations, as has previously been done for some *algorithmic skeletons*, such an approach requires excellent cost modelling, *and usually restricts the choice of implementation.* By using a refactoring approach, much of this machinery and the associated complexity can be avoided in favour of a programmer-directed system. The refactoring technology will also allow us to automatically insert appropriate component interfaces, including extra-functional (behavioural) information. In this way, we will help to achieve our overall aim of reducing the complexity of identifying parallelism for heterogeneous multicore systems.

Adaptive Mapping Technology (WP5). We need to develop methods to map software components onto the resources of a heterogeneous multicore platform, matching them against the available hardware characterisation that is exposed through the hardware/software virtualisation layers. This mapping needs to take into account both computations, which will mapped to the available

hardware resources, and any communication that is induced by this mapping. In this way, we will achieve our aim of developing new dynamic mechanisms to support adaptivity and heterogeneity.

Application-Based Validation (WP6). We have already identified a number of target high-performance applications from the data analysis, machine learning and weather prediction domains. These applications must demonstrate good multicore performance and expose opportunities for heterogeneity. They will be used to study the effectiveness of the various stages of our approach, including that the mapping and placement technology improves performance both in an initial placement onto a heterogeneous multicore application, and through system reconfiguration during execution.

User Community Building (WP7). ParaPhrase aims to create a user community to ensure longer-term uptake of the technologies developed in the project. Driven by the consortium industrial partners, this community will encompass a multiplicity of stakeholders exploiting close connections with the HPC Advisory Council (`http://www.hpcadvisorycouncil.com/`, "a computing ecosystem that includes best-in-class original equipment manufacturers (OEMs), strategic technology suppliers, independent software vendors (ISVs) and selected end-users across the entire range of HPC market segments."

3.4 ParaPhrase Use-Cases

The practical utilizability of the ParaPhrase approach especially concerning simplification of parallel development and performance gain will be demonstrated using real applications from industrial, scientific and video streaming domains. The focus on industrial applications for example is highly relevant in practice due to the trend to automation of manufacturing processes and machine control. Collected process data has the potential for optimizing those processes if analyzed in a proper way. But the complex relations, the huge amount of data and the often missing expertise make such optimizations hard to accomplish. Therefore sophisticated methods from the domain of machine learning (ML) and data mining (DM) are often required to identify the relations within the collected data. Furthermore high performance computational hardware is needed to perform these computations in a reasonable amount of time. As most ML algorithms currently only exist in a sequential version, easy transformation into parallel implementations is important as well. Solutions therefore require experts in machine learning for algorithm design and experts in parallelization for implementation on different hardware platforms. Different solutions to deal with the challenge of parallelization of ML algorithms on a higher level of abstraction have been proposed to cope with this problem. One approach is the adaption of the map-reduce (MR) paradigm to execute the algorithms on clusters or multicore machines [28] [58]. But this solution restricts the number of usable ML methods to those that fit the MR paradigm [16]. Other frameworks

like [48] have been published to overcome this restriction, but they do not exploit the potential of heterogeneous shared memory machines. With our use cases we want to demonstrate how the ParaPhrase approach can be used to overcome those shortcomings.

4 Conclusions

This paper has described the newly-started ParaPhrase project. It has introduced key technologies, described the structure of the project, the key related work in the area, and the advances that we anticipate making in the course of the project. ParaPhrase aims to mark a step change in programmability of heterogeneous parallel systems by synthesizing work from several independent areas and by developing new tools and techniques that will allow parallel programmers to develop programs using a tool-supported refactoring approach based on well-understood parallel design patterns coupled with good skeleton implementations. Each of the key technologies that will help achieve this goal is described in depth in a companion paper that has been submitted to this proceedings. Danelutto *et al.* describe a methodology suitable to implement autonomic management of multiple non functional concerns with the patterns and skeletons that we will use in the project; Brown *et al.* describe the refactoring tools and techniques; Gonzalez-Velez *et al.* describe the software virtualisations that we require; and, finally, Aldinucci *et al.* describe the hardware virtualisation layer that underpins the project. Achieving the overall goals of the project represents an exciting technical challenge that will require progress to be made in all these underlying technologies.

Acknowledgements. This work has been supported by the European Union Framework 7 grant IST-2011-288570 “ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems”, `http://www.paraphrase-ict.eu`.

References

1. Adve, S., Adve, V., Agha, G., Frank, M., et al.: Parallel@Illinois: Parallel Computing Research at Illinois — The UPCRC Agenda (November 2008), `http://www.upcrc.illinois.edu/documents/UPCRC_Whitepaper.pdf`
2. Aldinucci, M., Campa, S., Danelutto, M., Vanneschi, M., Dazzi, P., Laforenza, D., Tonellotto, N., Kilpatrick, P.: Behavioural skeletons in GCM: autonomic management of grid components. In: Euromicro PDP 2008, Toulouse, pp. 54–63. IEEE (February 2008)
3. Aldinucci, M., Danelutto, M., Dazzi, P.: MUSKEL: an expandable skeleton environment. Scalable Computing: Practice and Experience 8(4), 325–341 (2007)
4. Aldinucci, M., Danelutto, M., Kilpatrick, P.: Autonomic management of non-functional concerns in distributed and parallel application programming. In: IPDPS 2009, Rome, pp. 1–12. IEEE (May 2009)

5. Aldinucci, M., Danelutto, M., Kilpatrick, P.: Autonomic management of multiple non-functional concerns in behavioural skeletons. In: Desprez, F., Getov, V., Priol, T., Yahyapour, R. (eds.) Grids, P2P and Services Computing, pp. 89–103. Springer (2010)
6. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann Publishers (2001) ISBN 1-55860-286-0
7. Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K.: A view of the parallel computing landscape. Communications of the ACM 52(10), 56–67 (2009)
8. Bacon, D., Graham, S., Sharp, O.: Compiler Transformations for High-Performance Computing. ACM Computing Surveys 26(4), 345–420 (1994)
9. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques, Juan-les-Pins, France, pp. 7–16 (September 2004)
10. Bernecky, R., Herhut, S., Scholz, S.-B., Trojahner, K., Grelck, C., Shafarenko, A.: Index Vector Elimination – Making Index Vectors Affordable. In: Horváth, Z., Zsók, V., Butterfield, A. (eds.) IFL 2006. LNCS, vol. 4449, pp. 19–36. Springer, Heidelberg (2007)
11. Bharadwaj, V., Ghose, D., Mani, V., Robertazzi, T.G.: Scheduling Divisible Loads in Parallel and Distributed Systems. IEEE, Los Alamitos (1996)
12. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: PLDI 2008: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 101–113. ACM, New York (2008)
13. Buono, D., Danelutto, M., Lametti, S.: Map, reduce and mapreduce, the skeleton way. Procedia CS 1(1), 2095–2103 (2010)
14. Casanova, H., Kim, M.-H., Plank, J.S., Dongarra, J.: Adaptive scheduling for task farming with grid middleware. Int. J. High Perform. Comput. Appl. 13(3), 231–240 (1999)
15. Casavant, T., Kuhl, J.: A taxonomy of scheduling in general-purpose distributed computing systems. IEEE Trans. Softw. Eng. 14(2), 141–154 (1988)
16. Chu, C.-T., Kim, S.K., Lin, Y.-A., Ng, A.Y.: Map-reduce for machine learning on multicore. Architecture 19(23), 281 (2007)
17. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. Research Monographs in Parallel and Distributed Computing. Pitman/MIT Press, London (1989)
18. Danelutto, M.: On Skeletons and Design Patterns. In: Joubert, G.H., Murli, A., Peters, F.J., Vanneschi, M. (eds.) PARALLEL COMPUTING Advances and Current Issues Proceedings of the International Conference ParCo 2001. Imperial College Press (2002) ISBN: 1860943152
19. Danelutto, M.: HPC the easy way: new technologies for high performance application development and deployment. Journal of Systems Architecture 49(10-11), 399–419 (2003)
20. Dig, D.: A refactoring approach to parallelism. IEEE Softw. 28, 17–22 (2011)
21. El-Rewini, H., Lewis, T.G., Ali, H.H.: Task Scheduling in Parallel and Distributed Systems. Innovative Technology Series. Prentice Hall, New Jersey (1994)
22. Feautrier, P.: Some efficient solutions to the affine scheduling problem: I. one-dimensional time. Int. J. Parallel Program. 21(5), 313–348 (1992)
23. Feautrier, P.: Automatic Parallelization in the Polytope Model. In: Perrin, G.-R., Darte, A. (eds.) The Data Parallel Programming Model. LNCS, vol. 1132, pp. 79–103. Springer, Heidelberg (1996)

24. Eclipse Foundation: Eclipse - an Open Development Platform (2009), http://www.eclipse.org
25. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Abstraction and Reuse of Object-Oriented Design. In: Nierstrasz, O.M. (ed.) ECOOP 1993. LNCS, vol. 707, pp. 406–431. Springer, Heidelberg (1993)
26. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. Addison-Wesley, Upper Saddle River (1995)
27. Geppert, L., Perry, T.S.: Transmeta's magic show (microprocessor chips). IEEE Spectrum (2000)
28. Gillick, D., Faria, A., DeNero, J.: Mapreduce: Distributed computing for machine learning, Berkley (December 18, 2006)
29. González-Vélez, H., Cole, M.: An adaptive parallel pipeline pattern for grids. In: IPDPS 2008, Miami, USA, pp. 1–11. IEEE (April 2008)
30. González-Vélez, H., Cole, M.: Adaptive statistical scheduling of divisible workloads in heterogeneous systems. Journal of Scheduling 13(4), 427–441 (2010)
31. González-Vélez, H., Cole, M.: Adaptive structured parallelism for distributed heterogeneous architectures: A methodological approach with pipelines and farms. Concurrency and Computation–Practice & Experience 22(15), 2073–2094 (2010)
32. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. Software–Practice & Experience 40(12), 1135–1160 (2010)
33. Grelck, C.: Shared memory multiprocessor support for functional array processing in SAC. Journal of Functional Programming 15(3), 353–401 (2005)
34. Grelck, C., Hinckfuß, K., Scholz, S.-B.: With-Loop Fusion for Data Locality and Parallelism. In: Butterfield, A., Grelck, C., Huch, F. (eds.) IFL 2005. LNCS, vol. 4015, pp. 178–195. Springer, Heidelberg (2006)
35. Grelck, C., Kuthe, S., Scholz, S.-B.: A Hybrid Shared Memory Execution Model for a Data Parallel Language with I/O. Parallel Processing Letters 18(1), 23–37 (2008)
36. Grelck, C., Scholz, S.-B., Shafarenko, A.: A Binding Scope Analysis for Generic Programs on Arrays. In: Butterfield, A., Grelck, C., Huch, F. (eds.) IFL 2005. LNCS, vol. 4015, pp. 212–230. Springer, Heidelberg (2006)
37. Grelck, C., Scholz, S.-B., Trojahner, K.: With-Loop Scalarization – Merging Nested Array Operations. In: Trinder, P., Michaelson, G., Peña, R. (eds.) IFL 2003. LNCS, vol. 3145, pp. 118–134. Springer, Heidelberg (2004)
38. Griebl, M.: Automatic Parallelization of Loop Programs for Distributed Memory Architectures. University of Passau (2004) (Habilitation thesis)
39. Herhut, S., Scholz, S.-B., Bernecky, R., Grelck, C., Trojahner, K.: From Contracts Towards Dependent Types: Proofs by Partial Evaluation. In: Chitil, O., Horváth, Z., Zsók, V. (eds.) IFL 2007. LNCS, vol. 5083, pp. 254–273. Springer, Heidelberg (2008)
40. Kennedy, K., McKinley, K.S., Tseng, C.W.: Interactive parallel programming using the parascope editor. IEEE Trans. Parallel Distrib. Syst. 2, 329–341 (1991)
41. Kruskal, C., Weiss, A.: Allocating independent subtasks on parallel processors. IEEE Transactions on Software Engineering SE-11(10), 1001–1016 (1985)
42. Lengauer, C.: Loop Parallelization in the Polytope Model. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 398–416. Springer, Heidelberg (1993)
43. Leonard, P.: The Multi-Core Dilemma, Intel Software Blog (March 2007), http://software.intel.com/en-us/blogs/2007/03/14/the-multi-core-dilemma-by-patrick-leonard/

44. Lim, A.W., Lam, M.S.: Maximizing parallelism and minimizing synchronization with affine transforms - extended journal version parallel computing. Parallel Computing (1998)
45. Lim, A.W., Liao, S.-W., Lam, M.S.: Blocking and array contraction across arbitrarily nested loops using affine partitioning. In: PPoPP 2001: Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, pp. 103–112. ACM, New York (2001)
46. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Prentice Hall (1999)
47. Long, S., O'Boyle, M.: Adaptive java optimisation using instance-based learning. In: ICS 2004: Proceedings of the 18th Annual International Conference on Supercomputing, pp. 237–246. ACM, New York (2004)
48. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Graphlab: A new framework for parallel machine learning. CoRR, abs/1006.4990 (2010)
49. Majumdar, S., Eager, D.L., Bunt, R.B.: Scheduling in multiprogrammed parallel systems. SIGMETRICS Perform. Eval. Rev. 16(1), 104–113 (1988)
50. Mattson, T.G., Sanders, B.A., Massingill, B.L.: Patterns for Parallel Programming. Software Patterns Series. Addison-Wesley, Boston (2004)
51. Opdyke, W.F.: Refactoring object-oriented frameworks. PhD thesis, UIUC, Champaign, IL, USA (1992)
52. Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., Phillips, J.: GPU computing. Proceedings of the IEEE 96(5), 879–899 (2008)
53. Pop, A., Pop, S., Sjödin, J.: Automatic streamization in GCC. In: Proc. of the 2009 GCC Developers Summit, Montréal, Canada (June 2009)
54. Pouchet, L.-N., Bastoul, C., Cohen, A., Cavazos, J.: Iterative optimization in the polyhedral model: part ii, multidimensional time. In: PLDI 2008: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 90–100. ACM, New York (2008)
55. Rabhi, F.A., Gorlatch, S. (eds.): Patterns and Skeletons for Parallel and Distributed Computing. Springer, London (2003)
56. Scholz, S.-B.: With-Loop-Folding in Sac - Condensing Consecutive Array Operations. In: Clack, C., Hammond, K., Davie, T. (eds.) IFL 1997. LNCS, vol. 1467, pp. 72–91. Springer, Heidelberg (1998)
57. Shao, G., Berman, F., Wolski, R.: Master/slave computing on the grid. In: HCW 2000, Cancun, pp. 3–16. IEEE (May 2000)
58. Tamano, H., Nakadai, S., Araki, T.: Optimizing multiple machine learning jobs on mapreduce. In: 2011 IEEE Third International Conference on Cloud Computing Technology and Science, CloudCom, November 29-December 1, pp. 59–66 (2011)
59. Trojahner, K., Grelck, C., Scholz, S.-B.: On Optimising Shape-Generic Array Programs Using Symbolic Structural Information. In: Horváth, Z., Zsók, V., Butterfield, A. (eds.) IFL 2006. LNCS, vol. 4449, pp. 1–18. Springer, Heidelberg (2007)
60. University of Mannheim, University of Tennessee and NERSC. TOP500 supercomputer sites (November 2010), `http://www.top500.org/` (last accessed: December 1, 2010)
61. Vadhiyar, S.S., Dongarra, J.: Self adaptivity in grid computing. Concurr. Comput.-Pract. Exp. 17(2-4), 235–257 (2005)
62. Wloka, J., Sridharan, M., Tip, F.: Refactoring for reentrancy. In: ESEC/FSE 2009, pp. 173–182. ACM, Amsterdam (2009)
63. Wolfe, M.: High-Performance Compilers for Parallel Computing. Addison-Wesley (1995) ISBN 0-8053-2730-4

Paraphrasing: Generating Parallel Programs Using Refactoring

Christopher Brown[1], Kevin Hammond[1], Marco Danelutto[2], Peter Kilpatrick[3], Holger Schöner[4], and Tino Breddin[5]

[1] School of Computer Science, University of St. Andrews, Scotland KY16 9SX, UK
[2] Dept. Computer Science, Univ. Pisa, Largo Pontecorvo 3, 56127 PISA, Italy
[3] Sch. Electronics, Electrical Eng. and Comp. Sci., Queen's University Belfast, UK
[4] Software Competence Centre Hagenberg GmbH, Austria
[5] Erlang Solutions, London. UK
{chrisb,kh}@cs.st-andrews.ac.uk, marcod@di.unipi.it,
p.kilpatrick@qub.ac.uk, Holger.Schoener@scch.at

Abstract. Refactoring is the process of changing the structure of a program without changing its behaviour. Refactoring has so far only really been deployed effectively for sequential programs. However, with the increased availability of multicore (and, soon, *manycore*) systems, refactoring can play an important role in helping both expert and non-expert parallel programmers structure and implement their parallel programs. This paper describes the design of a new refactoring tool that is aimed at increasing the programmability of parallel systems. To motivate our design, we refactor a number of examples in C, C++ and Erlang into good parallel implementations, using a set of formal pattern rewrite rules.

1 Introduction

Despite Moore's "law" [22], uniprocessor clock speeds have now stalled. Rather than using single processors running at ever-higher clock speeds, and drawing ever-increasing amounts of power, even consumer laptops, tablets and desktops now have dual-, quad- or hexa-core processors. **Haswell**, Intel's next multicore architecture, will have eight cores by default. Future hardware is likely to have even more cores, with *manycore* and perhaps even *megacore* systems becoming mainstream. This means that programmers need to start *thinking parallel*, moving away from traditional programming models where parallelism is a bolted-on afterthought towards new models where parallelism is an intrinsic part of the software development process. One means of developing parallel programs that is attracting increasing interest is to employ parallel patterns, that is, sets of basic, predefined building blocks that each model and embed a frequently recurring pattern of parallel computation. An application is then a composition of these basic building blocks, that may be specialized by providing (suitably wrapped) sequential portions of code implementing the business logic of the application. Examples of such patterns include *farms*, *pipelines*, *map-reduces*, etc. By taking a pattern-based approach the application programmer can focus on providing

B. Beckert et al. (Eds.): FMCO 2011, LNCS 7542, pp. 237–256, 2013.
© Springer-Verlag Berlin Heidelberg 2013

the business code and, having identified a parallel pattern (composition) that is suitable for his/her application from a set of available patterns, can then get "for free" the necessary behind-the-scenes code, such as that for implementing synchronization and communication among the parallel activities. We thus envisage that an application programmer will begin with a sequential version of his/her business code and proceed to introduce parallelism by selecting parallel patterns that are suitable for the application at hand *and* for the target architecture. This, of course, requires expertise in pattern usage. It also requires suitable software support to facilitate introduction of patterns into the existing (sequential) code. This paper addresses this latter issue by exploring the use of refactoring as a means of bringing parallelism to business code via patterns.

1.1 Using Refactoring for Parallelism

Refactoring is the process of changing the internal structure of a program, while preserving its behaviour. The term *refactoring* was first introduced by Opdyke in 1992 [24], but the concept goes back to the fold/unfold system proposed by Darlington and Burstall in 1977 [10]. In contrast to general program transformations, such as *generic programming*, the key defining aspect of refactoring is its focus on *purely structural changes* rather than on changes in program functionality. Some advantages of refactoring are as follows:

– Refactoring aims to *improve software design*. Without refactoring, a program design will naturally decay: as code is changed, it progressively loses its structure, especially when this is done without fully understanding the original design. Regular refactoring helps tidy the code and retain its structure.
– Refactoring makes software *easier to understand*. Refactoring helps improve readability, and so makes code easier to change. A small amount of time spent refactoring means that the program better communicates its purpose.
– Refactoring helps the programmer to *program more rapidly*. Refactoring encourages good program design, which allows a development team to better understand their code. A good design is essential to maintaining rapid, but correct, software development.

Our refactoring tool will be developed as part of the ParaPhrase project, a new 3 year EU Framework-7 research project. ParaPhrase will use refactoring together with high-level design patterns[1] to introduce parallelism into sequential programs. In this paper we outline the design for the ParaPhrase refactoring tool that will refactor sequential programs written in C/C++ and Erlang to introduce parallelism, and will also refactor parallel programs in C/C++ and Erlang into more efficient implementations. By targeting C/C++ and Erlang we can demonstrate the effectiveness of our approach and its applicability to different paradigms.

[1] A *parallel (design) pattern* is a natural language description of a problem and of the associated solution techniques that the parallel programmer may use to solve that problem; an *algorithmic skeleton* is a programming entity used to implement a parallel design pattern. Here, for simplicity, we use the terms interchangeably.

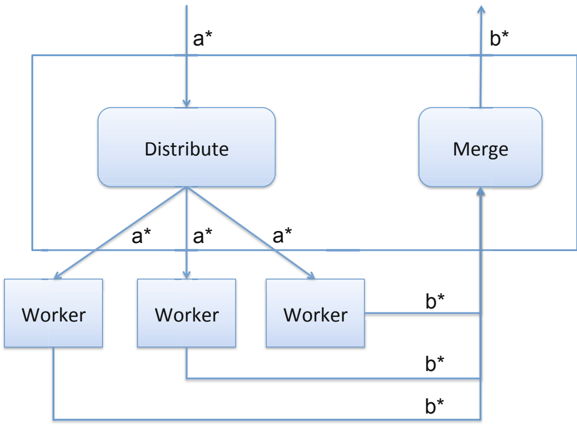


Fig. 1. A Typical Task Farm Showing a Master *Distribute* Function and the *Workers*

Listing 1. Sequential C Program Showing a Set of Tasks and a Worker Function *Before* the Refactoring Process

```
1  int main(int argc, char *argv[]) {
2    compute();
3  }
4  void compute() {
5      int i, task[MAX_TASKS];
6      for (i=0; i<MAX_TASKS; i++) {
7          task[i] = ... ;
8          payload(task[i]);  // set up some "tasks"
9      }
10 }
```

The specific technical contributions of this paper are:

1. we show how structured transformation techniques can enhance the *programmability* of parallel systems through refactoring;
2. we present a novel design for a new, generic, refactoring system that aims to transform programs into efficient parallel implementations, exploiting pattern-based rewrite rules that operate on systems of well-structured software components; and,
3. we present a number of new examples showing how refactoring can be used to aid a programmer in implementing parallelism.

Listing 2. Parallel C Program Showing an MPI Farm *After* the Refactoring Process

```
1  #include <mpi.h>
2  int main(int argc, char *argv[]) {
3    int np, rank;
4    MPI_Init(&argc, &argv);
5    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
6    MPI_Comm_size(MPI_COMM_WORLD,&np);
7    if (rank == 0) {
8       compute(np-1);
9    } else {
10      worker_component();
11   }
12   MPI_Finalize();
13 }
14 void compute(int workers) {
15     int i, task[MAX_TASKS];
16     for (i=0; i<MAX_TASKS; i++) {
17         task[i] = ... ;
18     }
19     for (i=0; i<workers; i++) {
20         MPI_Send(&task[i], 1, MPI_INT, i+1, i, MPI_COMM_WORLD);
21     }
22     while (i<MAX_TASKS) {
23       MPI_Recv(&temp, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
24       who = status.MPI_SOURCE;
25       tag = status.MPI_TAG;
26       result[tag] = temp;
27       MPI_Send(&task[i], 1, MPI_INT, who, i, MPI_COMM_WORLD);
28       i++;
29     }
30     for (i=0; i<workers; i++) {
31        MPI_Recv(&temp,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
32        who = status.MPI_SOURCE;
33        tag = status.MPI_TAG;
34        result[tag] = temp;
35        MPI_Send(&task[i], 1, MPI_INT, who, NO_MORE_TASKS, MPI_COMM_WORLD);
36     }
37 }
38 int computation(int x) {
39     return (payload(x));
40 }
41 void worker_component(){
42    int result, task;
43    MPI_Recv(&task, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
44    tag = status.MPI_TAG;
45    while (tag != NO_MORE_TASKS) {
46      result = computation(task);
47      MPI_Send(&result, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
48      MPI_Recv(&task, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD,&status);
49      tag = status.MPI_TAG;
50    }
51 }
```

2 Motivation

To motivate our refactoring design, we explore a simple refactoring example[2] that introduces a *task farm* [15] skeleton in C. A task farm is implemented as a series of *Worker* functions which are mapped to processor nodes. Each worker takes a set of *tasks* from a *master* and runs some computation that produces a *result* for each task. All results are fed back to the master. This is shown in Figure 1, where *Distribute* is the master, the sequence of inputs is shown as $a*$ and the sequence of results (which may be a different type to the tasks) is shown as $b*$. The *Merge* function merges the results as they are delivered.

The sequential program *before* the refactoring process is shown in Listing 1. The program itself is relatively simple: it simply creates some *tasks* and then performs a *computation* for each task. The computation itself is not important here, so we simply use the dummy function `payload`. In a real application, this function would be replaced by some meaningful computation for each task. The highlighted pieces of code are the parts that are needed as *inputs* to the refactoring tool. In the listing, the user has highlighted `compute` to act as the *Distribute* function; `task[i]` to represent the list of *Tasks* and `payload(task[i]);` to represent the *Worker* function. The refactorer will generate a *Merge* function, based on knowledge about C array processing. The user simply has to select these portions of code in a refactoring editor, and choose the *Introduce Task Farm* refactoring from the ParaPhrase refactorer. Preconditions, such as checking for non side-effecting code in the highlighted components would be done automatically, and the refactoring would fail if these conditions are not met. The refactored code is shown in Listing 2, which reveals the significant amount of boilerplate code that needs to be introduced to set up a task farm, including various low-level calls to MPI [27]. Most of the new code deals with accumulating the results and terminating the program. Once a worker processes a task, the result is returned to the master, and a new task is sent to the worker. When there are no more tasks, a termination message is sent to the worker. The important thing to note is that a refactoring tool will *automate all of these steps for the programmer*. The programmer can start with their sequential program, choose a *task farm* refactoring and have the refactoring tool produce the parallel version, complete with all the necessary MPI calls etc. For a complex program, this can be an *enormous* saving in effort. Even for a simple program, there is a significant saving in not needing to understand the detail of the MPI implementation.

In the remainder of this section, we demonstrate the *manual* steps that are needed to perform this refactoring by hand. We start with the simple C program shown in Listing 1. The first step in this refactoring process is to identify the *computation* component for the workers. In our example we identify the call to `payload` as the computation component and isolate this as a component:

```
1 int main(...) {
2 ...
3 }
4
```

[2] We use C and a task farm here for their familiarity and relative simplicity.

```
5  void compute() {
6      ...
7      for (i=0; i<MAX_TASKS; i++) { // set up some tasks
8          task[i] = ...
9          (void) computation(task[i]);
10     }
11 }
12
13 int computation(int x) {
14  return (payload(x));
15 }
```

The next stage is to identify the component that will represent the *workers* of the task farm. A refactoring tool will introduce this worker component automatically, by also introducing the MPI calls that send tasks to the workers.

```
1  int computation(int x);
2
3  int main(int argc, char *argv[]) {
4    int np, rank;
5    MPI_Status status;
6    MPI_Init(&argc, &argv);
7    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
8    MPI_Comm_size(MPI_COMM_WORLD, &np);
9
10   if (rank == 0)
11      compute(np-1);
12   else
13      worker_component();
14
15   MPI_Finalize();
16 }
17
18 void compute(int workers) {
19   int i, task[MAX_TASKS];
20   MPI_Status status;
21   for (i=0; i<MAX_TASKS; i++) { // set up some tasks
22          task[i] = ...
23          (void) computation(task[i]);
24   }
25
26   for (i=0; i<workers; i++)
27     MPI_Send(&task[i], 1, MPI_INT, i+1, i, MPI_COMM_WORLD);
28 }
29
30 void worker_component() {
31    int result, task;
32    MPI_Recv(&task, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
33    result = computation(task);
34 }
```

The `main` function is modified so that the workers are evaluated by separate MPI tasks. The main MPI task becomes the master (`compute`). Tasks are sent

Listing 3. Sequential Erlang Program *Before* Task Farm Refactoring

```
1 -module(taskFarm).
2 -export([run/2]).
3
4 run (Fun, Parameters) ->
5     [ Fun (P) || P <- Parameters ].
```

to the workers in a round-robin manner. At this stage the results are not yet accumulated, and no termination checking has been introduced. This is done in the final stage of the refactoring, which produces the code in Listing 2. Although there were only a small number of steps needed to perform this refactoring by hand, the programmer needs to understand how to implement a task farm skeleton, including checking for termination, and also needs expert knowledge in the use of MPI. It would be very easy to make a mistake at any of these steps, which could make debugging the parallel version very difficult. A refactoring tool, on the other hand, which automates this for a programmer, can eliminate potential mistakes and so allow the programmer to focus their effort on program design, rather than on the intricate details of implementing skeletons.

2.1 Erlang Example

We now show how to refactor an equivalent skeleton implementation of the same *task farm* in Erlang. Erlang [11] is a strict, impure, dynamically-typed functional programming language with support for higher-order functions, pattern matching, concurrency, communication, distribution, fault-tolerance and dynamic code loading. We begin with a sequential Erlang program, which simply maps a function Fun over a list, Parameters, as shown in Listing 3. Due to Erlang's functional style, functions are higher-order, meaning that they can take functions as arguments and return functions as results. We want to refactor this program into a task farm skeleton, as shown in Figure 1. The Erlang version is similar to the C version: we need to identify a number of components that will act as the *Workers* and the *Master*; a *Distribute* function is also required to merge the results of the workers. In Listing 3, the user has highlighted run as the *Master* component; Fun as the *Worker* and Parameters as the list of tasks.

The refactored version is shown in Listing 4. Here the refactoring has introduced a new function, do_run that takes three arguments: Fun, the computation to be performed by each worker; Parameter, the task sent to each worker; and Origin, the address of the master node on the network. Clearly, do_run acts as the *Worker* function in the task farm, computing the result by applying the computation Fun to the task, Parameter, and then sending the result back to the *Master*. Erlang uses the (!) primitive to send messages, and the receive primitive to receive messages. In the refactored example, the *Merge* function is expressed as the expression [receive {P, R} -> R end || P <- Procs]. which receives messages from the workers as they arrive. This *list comprehension* ensures that the merge only waits for the same number of results as there were original tasks. This merged list is then returned as the result of the program. It is also important to stress

Listing 4. Parallel Erlang Program *After* Task Farm Refactoring

```
1  -module(taskFarm).
2
3  -export([run/2, do_run/3]).
4
5  run(Fun, Parameters) ->
6      Procs = [spawn(?MODULE, do_run, [Fun, P, self()]) || P <- Parameters],
7      [receive {P, R} -> R end || P <- Procs].
8
9  do_run(Fun, Parameter, Origin) ->
10     Result = Fun(Parameter),
11     Origin ! {self(), Result}.
```

Listing 5. Parallel Erlang Program *After* a Renaming

```
1  -module(taskFarm).
2
3  -export([run/2, do_run/3]).
4
5  run(Fun, Tasks) ->
6      Procs = [spawn(?MODULE, do_run, [Fun, T, self()]) || T <- Tasks],
7      [receive {T, R} -> R end || T <- Procs].
8
9  do_run(Fun, Task, Origin) ->
10     Result = Fun(Task),
11     Origin ! {self(), Result}.
```

that the program in Listing 4 can undergo a further *renaming* refactoring by renaming Parameter in do_run to Task, P to T, and Parameters in run to Tasks. The completed code is shown in Listing 5.

3 The Design of the ParaPhrase Refactoring Tool

When constructing a refactoring tool, there are two main activities to consider: *program analysis* and *program transformation*. Program analysis checks whether certain side-conditions, which are necessary for the refactoring, are met and also collects any information that is needed during the program transformation phase. Program transformation performs the actual structural code changes that comprise a given refactoring. Both these steps are highly amenable to automation. The ParaPhrase refactorer will be *syntax independent*, initially working over C/C++ and Erlang. This demonstrates the generality of our approach, allowing patterns and rewrite rules to be expressed in terms of components rather than low-level language syntax. Targeting Erlang in addition to C/C++ also allows us to explore the advantages and limitations of both the imperative and functional paradigms, whilst also contributing to both user domains. Figure 2 shows

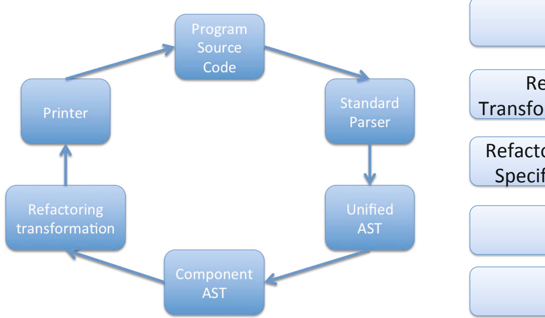


Fig. 2. The ParaPhrase Refactorer

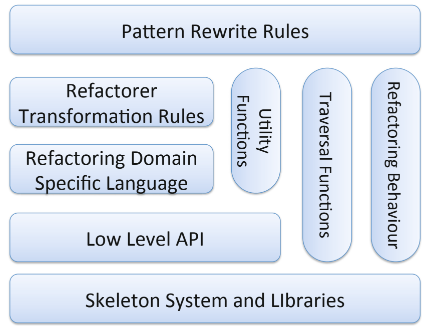


Fig. 3. The ParaPhrase Refactorer API

the design of the ParaPhrase refactorer, for C,C++ and Erlang. We note a number of components of the workflow in Figure 2:

1. The source syntax is parsed into a corresponding *Abstract Syntax Tree* (AST). Suitable static semantics must also be represented, such as use- and bind-locations for variables and the types of variables and functions. These static semantics are vital in order to correctly apply refactorings that make use of (and transform) the binding structure of a program, for example.
2. The AST is transformed into a *unified AST*, which must be general enough to express concepts from different paradigms and languages, yet still offer a sound representation of each syntax in order to transform and query it.
3. The unified AST is transformed into a *component AST* which can express the high level constructs in the program source that are required as arguments to the parallel patterns. Typically these constructs will be expressed as software components that might be identified automatically.
4. The component AST is refactored with respect to a set of well-defined transformation rules for the parallel patterns. The output of this refactoring will often be a modified version of the unified AST.
5. The refactored component AST is "pretty-printed" in the source syntax. Layout and comments should be preserved where possible, so that the programmer is presented with a refactored program that preserves their programming style and idiom.

The ParaPhrase refactoring tool will be made *user-extensible* through a number of layers of API abstractions, as shown in Figure 3. We predict that the following will be needed to support user-level pattern-based refactorings:

1. Patterns will be expressed in a high level abstract language that will allow users to write rules to *introduce* and *eliminate* patterns, together with their *composition*. This pattern language will be void of any syntax information and will be general enough to express pattern rewrites for all syntaxes. These rules are described in more detail in Section 3.1.
2. A language for expressing refactoring transformations will allow the refactorings themselves to be expressed in terms of a general syntax, including pre- and post-conditions and transformation rules.

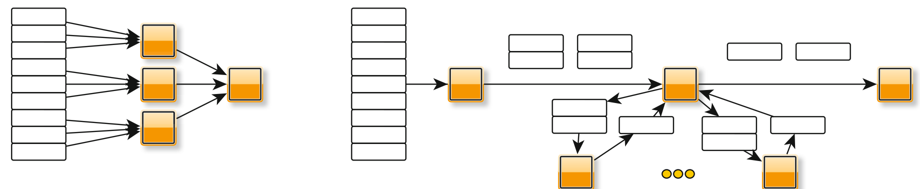


Fig. 4. Map-Reduce pattern (left) and an equivalent Pipeline-Farm pattern (right)

3. A refactoring Domain Specific Language (DSL) framework will allow for the composition of the refactorings to form larger refactorings.
4. A collection of *utility* functions such as traversal functions for the Abstract Syntax Trees, retrieval of binding information, mapping an editor selection onto its Syntax Tree representation, etc.
5. A Skeleton library that the refactored source program can *import* to access low-level skeleton implementation details.
6. A concrete interface that will be integrated into a popular editing environment, such as Emacs or Eclipse.

Since there is already a refactoring tool for Erlang, Wrangler [17], that uses an expressive Domain Specific Language (DSL) to define refactorings in terms of their pre-conditions and transformation rules as Erlang macros, when dealing with Erlang it may be possible to plug our pattern-rewrite rules directly into the Wrangler DSL rather than using our own generic refactorer.

3.1 Patterns as Rewrite Rules

The ParaPhrase approach is based around the use of parallel patterns to drive the program transformations in the refactoring tool. Parallel patterns impose a clear and easily-recognised structure on the forms of parallelism that can be exploited in an application. As an example, if we use the well-known *map-reduce* pattern (Figure 4) to model parallel behaviour, then:

1. the signature of the function used to transform all the data collection items during the map phase is known;
2. the signature of the function used to "sum up" all the items in the result collection is known;
3. the data dependencies are known;
4. and it may, perhaps, be known whether the reduce operator is both associative and commutative.

All this information may be used to refactor the parallel computation in terms of other parallel patterns. The *map-reduce* computation could be expressed, for example, in terms of a *pipeline* pattern whose stages are:

- a stage splitting the input collection and delivering partitions of the collection to the next stage (*splitter* stage, sequential);
- a stage modelled after the "embarrassingly parallel" parallel pattern (the *partition map-reduce* stage, parallel), processing each partition by:

  - first applying the map operator to each partition item;
  - then applying the reduce operator to "sum up" all the computed results;
  - finally delivering the result to the stream leading to the next stage.
- a stage gathering the results from the partitions and summing them again, using the reduce operator.

The overall refactoring in this case transforms a composition of *map* and *reduce* patterns into a composition of *pipeline* and *farm* (the embarrassingly parallel pattern on streams) patterns, as shown in Figure 4. For a stream of input collections, the refactored program may exploit more parallelism (and therefore better performance). A number of "rewrite rules" can be used once parallel patterns are identified with all their functional and non-functional parameters. Assuming we have a pattern *palette* defined by the following BNF:

$$
\begin{array}{lcl}
Pattern & ::= & Pipe \mid Farm \mid Comp \mid Map \mid Reduce \mid Seq \\
Pipe & ::= & pipe(Pattern, Pattern) \\
Farm & ::= & farm(Pattern) \\
Comp & ::= & comp(Pattern, Pattern) \\
Map & ::= & map(Pattern) \\
Reduce & ::= & reduce(Pattern) \\
Seq & ::= & \langle sequential\ code\ wrapping \rangle
\end{array}
$$

then the rules in Table 1 can be used to apply "parallel" refactoring. All these rules preserve the "functional" semantics, but not the "parallel" semantics. Both sides of each rule give the same results, but the computation may involve different parallel patterns and therefore different performance. The performance obviously depends on various parameters, including application-related and target-architecture related ones. These rules are used *de facto* by the *Pattern Rewrite Rules* box in Figure 3 to drive the *refactorer*. It is worth pointing out that:

- where all the parallel patterns are directly available as algorithmic skeletons (e.g. as entries of a skeleton library), the refactoring process may simply consist of substituting sequences of library calls;

Table 1. Parallel pattern rewriting rules. The coll2singleton and singleton2coll functions transform a collection into a stream of collection item components and vice versa.

P	$\rightarrow$	farm(P)	farm introduction
farm(P)	$\rightarrow$	P	farm elimination
pipe(pipe(P1,P2),P3)	$\equiv$	pipe(P1,pipe(P2,P3))	pipeline assoc
pipe(map(P1),map(P2))	$\equiv$	map(pipe(P1,P2))	pipe/map distrib
pipe(map(P1),reduce(P2))	$\rightarrow$	pipe(comp(map(P1),reduce(P2)),reduce(P2))	map reduce promotion
map(P1)	$\equiv$	pipe(coll2singleton,farm(P1),singleton2coll)	map/farm equivalence
pipe(P1,P2)	$\rightarrow$	comp(P1,P2)	stream par elimination
comp(P1,P2)	$\rightarrow$	pipe(P1,P2)	stream par introduction

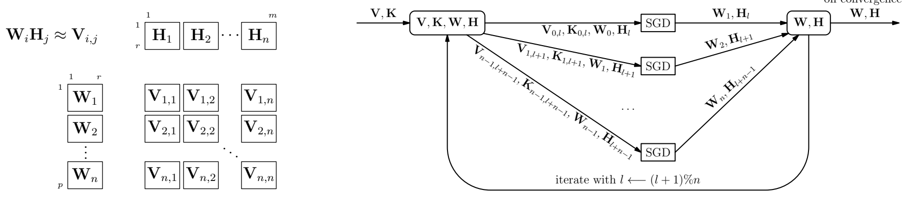


Fig. 5. Structure of coarse parallelization approach. By dividing matrix $\mathbf{V}$ into blocks, only parts i and j of matrices $\mathbf{W}$, $\mathbf{H}$, respectively, are used for the approximation of each block (i, j) of $\mathbf{V}$. n blocks can thus be processed simultaneously with Stochastic Gradient Descent (SGD) without conflicts of the parameter updates. After some SGD iterations, the parameters $\mathbf{W}$, $\mathbf{H}$ are collected again, and redistributed using a different set of n non-overlapping blocks of $\mathbf{V}$. As $\mathbf{V}$ has $n \times n$ different blocks, there are n iterations of n parallel SGD updates until one sweep through all values in $\mathbf{V}$ is completed (one epoch of an equivalent global SGD). $\mathbf{K}$ is a $k \times 2$ matrix containing in each row the index into V of one of the k known values of $\mathbf{V}$, needed to compute the SGD iterations.

– even where skeletons are available, the refactoring using the rules above may require new sequential code to be generated. For example, for the task farm refactoring above, the *worker* code for the second stage pipeline requires some specific code to iterate the map operator over all the elements of the partition. In most cases (e.g. in the rules of Table 1) the refactoring only requires changing the (sequence of) calls to the skeleton library.

4 Use Case: Large Scale Matrix Factorization

Matrix Factorization for Data Modeling. Factorization of large matrices is a method common in applications like recommender systems, and user preference modeling [16]. An example is a problem setting in which partial data of the ratings of a large number of persons for a large number of movies are considered [1]. This problem setting provides a suitable vehicle for parallelization and refactoring, as the approach and its parallelization are simple enough for presentation here, while still being of real world significance. In the following development we omit problem specific sequential code (cf. [13]), and concentrate on the parallelization and refactoring.

The known and unknown ratings of users for movies are collected in a rating matrix $\mathbf{V}$ with size $p \times m$. The rows correspond to persons, and the columns to movies. Most of the values in this matrix will be unknown, as usually users rate only a limited number of movies. An auxiliary $k \times 2$ matrix $\mathbf{K}$ is used to keep track of the known values of $\mathbf{V}$: row $(s, t) \in \mathbf{K} \Leftrightarrow \mathbf{V}_{s,t}$ is known.

To obtain estimates for probable ratings of users for movies they have not rated (or even seen) so far, a factorization of $\mathbf{V}$ into the $p \times r$ row factor matrix $\mathbf{W}$ and the $r \times m$ matrix $\mathbf{H}$ is searched, with $\mathbf{V} \approx \mathbf{WH}$ (for the known entries in $\mathbf{V}$). Estimate of an unknown rating of person s for movie t is the dot product $\mathbf{W}_{s\cdot}\mathbf{H}_{\cdot t}$.

r is the number of factors, and $\mathbf{W}$ and $\mathbf{H}$ are the factor memberships of the persons and movies. Factors could correspond to groups of movies and persons, like action movies and action loving persons, or romantic movies/persons.

The factorization is performed by optimization of a cost function w.r.t. the learned parameters $\mathbf{W}$, $\mathbf{H}$. The cost function is the mean of some loss over the known rating values, e.g. the squared difference of the rating estimates to the real ratings, for all known values $(s, t) \in \mathbf{K}$. The optimization is performed by gradient descent, which iterates epochs (one sweep through all known values) of learning until convergence of the cost value.

Parallelization Approach. This problem can be parallelized on two levels. The general approach for a high level parallelization is sketched in Figure 5. The matrix $\mathbf{V}$ is split into blocks, the parameter matrices into stripes. The matrix factorization can now be performed for each block of $\mathbf{V}$ separately, and all blocks corresponding to independent stripes of $\mathbf{W}$ and $\mathbf{H}$ can be optimized in parallel. As blocks of $\mathbf{V}$ in the same row or column share parameters and cannot be optimized in parallel, one epoch of learning needs an additional loop around these parallel optimizations, until all blocks are optimized once. Afterwards, the whole process is repeated until convergence of the parameters. The square root n of the number of blocks $n \times n$ of matrix $\mathbf{V}$ is the degree of parallelization, and can be tuned to the number of processing elements, and to the problem size.

A sequential version of this approach, comprising the starting point of the following refactoring, is shown in the C++-like pseudo code in Listing 6. The details of functions `stochasticGradientDescent`, `partitionMatrices` and `randomMatrix` are not important for the high level parallelization, and the implementation is just sketched. The second level of parallelization can be performed inside the stochastic gradient descent. The computations of stochastic gradient descent are mainly linear algebra. They are well parallelizable using a data-parallel approach. ParaPhrase will also provide appropriate patterns, e.g. Map-Reduce, and support for heterogeneous parallel architectures, such as distributed multicore systems with GPGPUs on each node. The matrix factorization example could then be mapped onto such a cluster by distributing the matrix blocks to the available computation nodes, and parallelizing the stochastic gradient descent on their respective GPGPUs.

Parallelization by Refactoring. Listing 6 is the starting point of refactoring for parallelization. We wish to use the *Farm* pattern to distribute the n parallel SGD computations. Before the *Farm* pattern can be applied, the worker function performing the SGD needs to be wrapped inside a *Seq* pattern, as other refactoring rules can only be applied to existing patterns (cf. Section 3.1). This is achieved by using the refactorer to mark the `stochasticGradientDescent` function and wrap it in a *Seq* pattern. Afterwards, the "P $\rightarrow$ farm(P)" rewriting rule can be used, to make the sequential loop parallel. The code which results from those refactorings is given in Listing 7, with the changes highlighted. Similar refactorings are also possible to introduce *map-reduce* parallelism in the linear algebra operations performed by the stochastic gradient. The convenience of code rewriting is one advantage of automatic refactoring; in addition, the following considerations make it a valuable tool for applications such as the one above:

Listing 6. Sequential version of matrix factorization

```
1  (Matrix, Matrix) matrixFactorization(Matrix V, Matrix K, int r, int n) {
2    // initializations ...
3    List<List<Matrix>> blocksV, filteredK;
4    List<Matrix> rowsW, colsH;
5    // assume that blocksV, ... are views of parts of V, ...,
6    // such that updates to rowsW, colsH also update W, H
7    (blocksV, filteredK, rowsW, colsH) = partitionMatrices(n, V, K, W, H);
8    while (!converged(loss, V, K, W, H)) {
9      loss = 0;
10     for (l ∈ {0, ..., n − 1}) {
11       for (i ∈ {0, ..., n − 1}) {
12         int j = (l+i) % n;
13         (loss_part, rowsW[i], colsH[j]) = stochasticGradientDescent(
14           blocksV[i,j], filteredK[i,j], rowsW[i], colsH[j]);
15         loss += loss_part;
16       }
17     }
18     loss /= n;
19   }
20   return (W, H);
21 }
22
23 // Performs one epoch of Stochastic gradient descent, returning loss and new W, H.
24 // An epoch corresponds to one sweep over all (known) elements of the matrix
25 (double, Matrix, Matrix) stochasticGradientDescent(Matrix V, Matrix K,
26     Matrix W, Matrix H) {
27   // ...
28 }
29
30 // split V, K, W, H into n partitions (n × n for V and K).
31 (List<List<Matrix>>, List<List<Matrix>>, List<Matrix>, List<Matrix>)
32     partitionMatrices(int n, Matrix V, Matrix K, Matrix W, Matrix H) {
33   // ...
34   // filteredK[i][j] contains all rows u of K, for which K[u] = (s, t) is
35   // an index into block (i, j) of V, recomputed to be the
36   // equivalent index into blocksV[i][j]
37   return (blocksV, filteredK, rowsW, colsH);
38 }
```

- checks will be performed to determine whether the intended refactoring is possible, and whether there are conflicts on the new identifiers;
- switches between different kinds of parallel patterns will be easier, facilitating the optimization of the patterns used for the application and for the available hardware; and
- guidance might be available regarding an optimal choice of patterns given non-functional criteria such as run-time considerations.

Listing 7. *Farm* version of matrix factorization

```
1  (Matrix, Matrix) matrixFactorization(Matrix V, Matrix K, int r, int n) {
2    // ...
3    (blocksV, filteredK, rowsW, colsH) = partitionMatrices(n, V, K, W, H);
4    Pattern seqSGD = SequentialPattern(stochasticGradientDescent);
5    Pattern farmSGDEpoch = FarmPattern(seqSGD);
6    while (!converged(loss, V, K, W, H)) {
7      loss = 0;
8      for (l ∈ {0, ..., n − 1}) {
9        for (i ∈ {0, ..., n − 1}) {
10         int j = (l+i) % n;
11         farmSGDEpoch.execute(blocksV[i,j], filterdK[i,j], rowsW[i], colsH[j]);
12       }
13       List<(double,Matrix,Matrix)> resList = farmSGDEpoch.waitforall();
14       for (i ∈ {0, ..., n − 1}) {
15         int j = (l+i) % n;
16         loss_part = resList[i][0]; rowsW[i] = resList[i][1]; colsH[j] = resList[i][2];
17         loss += loss_part;
18       }
19     }
20     loss /= n;
21   }
22   return (W, H);
23 }
```

Pattern Refactoring. To exploit the ease of refactoring parallel patterns, a refactoring to use a parallel *Map* pattern instead of a *Farm* pattern is now considered. Such refactoring could be useful for clarifying the program structure, or it might be more appropriate for a given parallel architecture. Automatic Refactoring allows easy switching between such patterns, making it straightforward to explore the available alternatives and find the optimal one for a given situation. To apply the *Map* pattern, the input arguments for the *Farm* workers have to be collected in a list, automatized as much as possible by the refactoring, to which the already existing *Seq* pattern can then be applied by the *Map*, as shown in Listing 8 with refactoring changes highlighted.

Refactoring Analysis. Comparing the three versions of the matrixFactorization function, it is obvious that large parts are very similar, a prerequisite for automatic refactoring. Still, it is also obvious that refactoring does not mean just a simple exchange of a FarmPattern by a MapPattern, or similar. Parts of the surrounding code have to be reorganized as well. In this example, the following issues arise:

Listing 8. *Map* version of matrix factorization

```
1  (Matrix, Matrix) matrixFactorization(Matrix V, Matrix K, int r, int n) {
2    // ...
3    (blocksV, filteredK, rowsW, colsH) = partitionMatrices(n, V, K, W, H);
4    Pattern seqSGD = SequentialPattern(stochasticGradientDescent);
5    Pattern mapSGDEpoch = MapPattern(seqSGD);
6    while (!converged(loss, V, K, W, H)) {
7      loss = 0;
8      for (l ∈ {0,...,n−1}) {
9        List<(Matrix,Matrix,Matrix,Matrix)> mapList;
10       for (i ∈ {0,...,n−1}) {
11         int j = (l+i) % n;
12         mapList.append( (blocksV[i,j], filteredK[i,j], rowsW[i], colsH[j]) );
13       }
14       List<(double,Matrix,Matrix)> resList = mapSGDEpoch.execute(mapList);
15       for (i ∈ {0,...,n−1}) {
16         int j = (l+i) % n;
17         loss_part = resList[i][0];  rowsW[i] = resList[i][1]; colsH[j] = resList[i][2];
18         loss += loss_part;
19       }
20     }
21     loss /= n;
22   }
23   return (W, H);
24 }
```

- introducing new variables; e.g. the variables for the pattern instances, but also those collecting the *Farm* or *Map* results;
- handling arguments of the original worker function and its results, e.g. by wrapping and unwrapping to and from single list items;
- splitting of loops, such that results of several iterations can be collected outside the loop, while unwrapping the results might necessitate replicating the loop a second time and repeating parts of the loop (here, "**int** j=(l+i)%n" is necessary a second time);
- here, the sequential version already contained the code necessary for splitting the four distributed matrices into blocks; it would be desirable, although maybe not realistic, to also have the refactoring introduce such helper functions as necessary;
- the sequential version has already ensured that the worker function has no side effects; it might be desirable to introduce refactorings transforming functions (and calls to them) which are not yet side-effect free.

How many of these tasks can be performed automatically, and what other tasks might be necessary for other use cases, remains to be investigated during the ParaPhrase project.

5 Related Work

Program transformation has a long history, with early work in the field being described by Partsch and Steinbruggen in 1983 [26] and Mens and Tourwé producing an extensive survey of refactoring tools and techniques in 2004 [21]. The first refactoring tool system was the *fold/unfold* system of Burstall and Darlington [10] which was intended to transform recursively defined functions. The overall aim of the *fold/unfold* system was to help programmers to write correct programs which are easy to modify. There are six basic transformation rules that the system is based on: unfolding; folding; instantiation; abstraction; definition and laws. The advantage of using this methodology was that it deployed a number of simple, and yet effective, structural program transformations that aimed to develop more efficient definitions; the disadvantage was that the use of the fold rule sometimes resulted in non-terminating definitions.

The Haskell Refactorer, HaRe, is a semi-automated refactoring tool for sequential Haskell programs, developed at the University of Kent by Thompson, Li and Brown [8]. HaRe works over the full Haskell 98 standard, and contains a large catalog of refactorings that concentrate on small structural changes in sequential Haskell programs, such as *renaming*, *lambda lifting* and type-based refactorings. HaRe was recently extended by us to deal with a limited number of parallel refactorings. This technique is known as *paraforming* [9], and allows Haskell programmers to construct data and task parallelism using small structural refactoring steps, although it does not use pattern-based rewriting, as in ParaPhrase. Wrangler [17], also developed at the University of Kent by Thompson and Li, is similar to HaRe, but works over sequential Erlang instead. Wrangler also contains a large database of refactorings for Erlang programs and includes a Domain Specific Language for expressing transformations (together with their conditions) [19] and a further language for composing refactorings [18]. Unlike our requirements for the ParaPhrase project, Wrangler does not deal with parallel refactorings in any way. Cocinelle and its DSL framework, SmPL [25], is a program matching and transformation engine for specifying desired matches and transformations in C code. Stratego/XT [7] provides a language-independent framework for expressing refactorings over arbitrarily defined syntaxes. It may be possible to use Stratego in the context of Paraphrase for expressing the parallel refactorings.

There has been a limited amount of work on parallel refactoring in general, mostly with loop parallelisation in Fortran [29] and Java [12]. However these approaches are limited to concrete structural changes (such as loop unrolling) rather than applying high-level pattern-based rewrites. A companion paper contains a much more detailed survey of refactoring tools for parallelisation [14].

Rewriting of structured parallel programs has been studied in different contexts. Backus' Turing award lecture note [4], although not explicitly dealing with parallel patterns or design patterns, sets up a scenario that allows to "compute" program transformations using an algebra of programs. In particular, several transformations related to map and reduce second order functions are already present in this work. As an example the "pipe/map" rule described in Table 1 was already present in that seminal work described as

$$\alpha f \circ \alpha g \equiv \alpha(f \circ g)$$

with the functional composition operator $\circ$ representing pipeline, as usual (stream parallel), and the *apply-to-all* α representing the map skeleton. More recently more rewriting rules have been designed for algorithmic skeletons. In [2] a concept of "normal form" for stream parallel skeleton compositions is introduced that maximizes the service time by applying systematic rewriting of arbitrary stream parallel skeleton compositions to farms with sequential workers. Other authors consider usage of different skeleton rewriting rules to target different architectures, in particular those including GPUs [23]. The full SkeTo skeleton based programming framework (see `http://sketo.ipl-lab.org/`) uses the results from Bird-Merteens theory [6,28] to optimize data parallel skeleton compositions [20]. The potential refactoring for C++ code by introducing algorithmic skeletons has been previously discussed in the context of FastFlow (see `http://calvados.di.unipi.it/dokuwiki/doku.php?id=ffnamespace:about`). In particular, in [3] the refactoring of sequential code to introduce a task farm skeleton is discussed. Finally, PEPPHER: *Performance Portability and Programmability for Heterogeneous Many-core Architectures* [5], is an EU FP7 funded project that started in January 2010. The aim of PEPPHER is to devise a unified framework for programming and optimizing applications for a diverse range of architectures such as heterogeneous many-core processes in order to ensure performance portability. PEPPHER uses direct compilation to the target architectures, so portability is supported by powerful composition methods with a toolbox of adaptive algorithms. However, the PEPPHER project does not use refactoring and pattern rewriting to increase the programmability of parallel systems as we do here, relying instead on adaptive algorithms and architecture-directed compilation.

6 Conclusions

This paper has described a new design methodology for the ParaPhrase refactoring tool, a radically new system that will refactor systems of software components into efficient parallel implementations. The tool will refactor programs written in C, C++ and Erlang (although we also expect to extend our tool to deal with other languages such as Haskell and Python), by applying high-level abstract pattern rewrite rules that can either:

- Introduce new patterns into an existing sequential program; or,
- Modify an existing parallel program by changing the pattern already specified, or introducing a new pattern, therefore composing patterns together.

A companion paper by Hammond *et al.* [14] gives an overview of the ParaPhrase project encompassing many key technologies and techniques that we will employ in addition to refactoring. Among these is the need to identify means of specifying non-functional properties of systems in such a way that it becomes possible to verify that a refactoring achieves its intended purpose. We have demonstrated the effectiveness of having such a refactoring tool by motivating our design with a number of key examples in C, Erlang and C++, where we showed that having a refactoring tool can effectively *automate* the majority of the boilerplate implementation detail of introducing a new skeleton,

such as adding MPI code in C, or adding a master/worker skeleton in Erlang. This is potentially an *enormous* saving in effort, allowing the programmer to focus on designing algorithms rather than worrying about the details of parallel implementation. We believe this is the correct way to improve substantially the *programmability* of such parallel systems. A refactoring tool such as the one described here would be a key component in increasing the productivity of parallel programming in general.

Acknowledgments. This work has been supported by the European Union grants RII3-CT-2005- 026133 SCIEnce: Symbolic Computing Infrastructure in Europe, IST-2010- 248828 ADVANCE: Asynchronous and Dynamic Virtualisation through performance ANalysis to support Concurrency Engineering, and IST-2011-288570 ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems, and by the UK's Engineering and Physical Sciences Research Council grant EP/G055181/1 HPC-GAP: High Performance Computational Algebra.

References

1. Netflix Prize Forum/Grand Prize Award (September 2009), `http://www.netflixprize.com/community/viewtopic.php?id=1537`
2. Aldinucci, M., Danelutto, M.: Stream Parallel Skeleton Optimization. In: Proc. of PDCS: Intl. Conference on Parallel and Distributed Computing and Systems, pp. 955–962. IASTED, ACTA Press, Cambridge, Massachusetts (1999)
3. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: Accelerating Code on Multi-cores with FastFlow. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part II. LNCS, vol. 6853, pp. 170–181. Springer, Heidelberg (2011)
4. Backus, J.: Can Programming be Liberated from the von Neumann Style? Communications of the ACM 21(8), 613–641 (1978)
5. Benkner, S., Pllana, S., Träff, J.L., Tsigas, P., Dolinsky, U., Augonnet, C., Bachmayer, B., Kessler, C.W., Moloney, D., Osipov, V.: PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems. IEEE Micro 31(5), 28–41 (2011)
6. Bird, R.S.: Lectures on Constructive Functional Programming. In: Broy, M. (ed.) Constructive Methods in Computer Science. NATO ASI Series F, vol. 55, pp. 151–218. Springer (1988); Also available as Technical Monograph PRG-69, from the Programming Research Group, Oxford University
7. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/xt 0.17. A Language and Toolset for Program Transformation. Sci. Comput. Program. 72(1-2), 52–70 (2008)
8. Brown, C., Li, H., Thompson, S.: An Expression Processor: A Case Study in Refactoring Haskell Programs. In: Page, R., Horváth, Z., Zsók, V. (eds.) TFP 2010. LNCS, vol. 6546, pp. 31–49. Springer, Heidelberg (2011)
9. Brown, C., Loidl, H.-W., Hammond, K.: ParaForming: Forming Parallel Haskell Programs Using Novel Refactoring Techniques. In: Peña, R., Page, R. (eds.) TFP 2011. LNCS, vol. 7193, pp. 82–97. Springer, Heidelberg (2012)
10. Burstall, R.M., Darlington, J.: A Transformation System for Developing Recursive Programs. J. ACM 24(1), 44–67 (1977)
11. Cesarini, F., Thompson, S.: ERLANG Programming, 1st edn. O'Reilly Media, Inc. (2009)

12. Dig, D.: A Refactoring Approach to Parallelism. IEEE Softw. 28, 17–22 (2011)
13. Gemulla, R., Haas, P.J., Sismanis, Y., Teflioudi, C., Makari, F.: Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent. In: NIPS 2011 Workshop on Big Learning, Sierra Nevada, Spain (December 2011)
14. Hammond, K., Aldinucci, M., Brown, C., Cesarini, F., Danelutto, M., González-Vélez, H., Kilpatrick, P., Keller, R., Rossbory, M., Shainer, G.: The ParaPhrase: Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems. In: Beckert, B., de Boer, F., Bonsangue, M., Damiani, F. (eds.) FMCO 2011. LNCS, vol. 7542, pp. 218–236. Springer, Heidelberg (2012)
15. Klusik, U., Loogen, R., Priebe, S., Rubio, F.: Implementation Skeletons in Eden: Low-Effort Parallel Programming. In: Mohnen, M., Koopman, P. (eds.) IFL 2000. LNCS, vol. 2011, pp. 71–88. Springer, Heidelberg (2001)
16. Koren, Y., Bell, R., Volinsky, C.: Matrix Factorization Techniques for Recommender Systems. IEEE Computer 42(8), 30–37 (2009)
17. Li, H., Thompson, S.: A Comparative Study of Refactoring Haskell and Erlang Programs. In: SCAM 2006, pp. 197–206. IEEE (September 2006)
18. Li, H., Thompson, S.: A Domain-Specific Language for Scripting Refactorings in Erlang. Technical Report 5-11, University of Kent (October 2011)
19. Li, H., Thompson, S.: A User-extensible Refactoring Tool for Erlang Programs. Technical Report 4-11, University of Kent (October 2011)
20. Matsuzaki, K., Iwasaki, H., Emoto, K., Hu, Z.: A Library of Constructive Skeletons for Sequential Style of Parallel Programming. In: Proceedings of the 1st International Conference on Scalable Information Systems, InfoScale 2006. ACM, New York (2006)
21. Mens, T., Tourwé, T.: A Survey of Software Refactoring. IEEE Trans. Softw. Eng. 30(2), 126–139 (2004)
22. Moore, G.E.: Cramming more components onto integrated circuits. In: Readings in Computer Architecture, pp. 56–59. Morgan Kaufmann Publishers Inc., San Francisco (2000)
23. Nugteren, C., Corporaal, H., Mesman, B.: Skeleton-based Automatic Parallelization of Image Processing Algorithms for GPUs. In: ICSAMOS 2011, pp. 25–32 (2011)
24. Opdyke, W.F.: Refactoring Object-Oriented Frameworks. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Champaign, IL, USA (1992)
25. Padioleau, Y., Lawall, J.L., Muller, G.: SmPL: A Domain-Specific Language for Specifying Collateral evolutions in Linux device drivers. In: International ERCIM Workshop on Software Evolution, Lille, France (April 2006)
26. Partsch, H., Steinbruggen, R.: Program Transformation Systems. ACM Comput. Surv. 15(3), 199–236 (1983)
27. Sankaran, S., Squyres, J.M., Barrett, B., Lumsdaine, A., Duell, J., Hargrove, P., Roman, E.: The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. International Journal of High Performance Computing Applications 19(4), 479–493 (2005)
28. Skillicorn, D.B.: The Bird-Meertens Formalism as a Parallel Model. In: Software for Parallel Computation. NATO ASI Series F, vol. 106, pp. 120–133. Springer (1993)
29. Wloka, J., Sridharan, M., Tip, F.: Refactoring for Reentrancy. In: ESEC/FSE 2009, pp. 173–182. ACM, Amsterdam (2009)

An Abstract Annotation Model for Skeletons

Marco Aldinucci[1], Sonia Campa[2], Peter Kilpatrick[3], Fabio Tordini[1], and Massimo Torquati[2]

[1] Computer Science Department, University of Torino, Italy
{aldinuc,fabio.tordini}@di.unito.it
[2] Computer Science Department, University of Pisa, Italy
{campa,torquati}@di.unipi.it
[3] Computer Science Department, Queen's University Belfast, UK
p.kilpatrick@qub.ac.uk

Abstract. Multi-core and many-core platforms are becoming increasingly heterogeneous and asymmetric. This significantly increases the porting and tuning effort required for parallel codes, which in turn often leads to a growing gap between peak machine power and actual application performance. In this work a first step toward the automated optimization of high level skeleton-based parallel code is discussed. The paper presents an abstract annotation model for skeleton programs aimed at formally describing suitable mapping of parallel activities on a high-level platform representation. The derived mapping and scheduling strategies are used to generate optimized run-time code.

1 Introduction

One central challenge of parallel programming today is to achieve performance portability across a range of architectures. Most application programs are currently written at the low level of C or Fortran, combined with a communication library such as MPI; moreover, they are often tuned toward one specific machine configuration. Since parallel computers are typically replaced within five years, parallel programs which have a longer life span have to be re-tuned or redesigned. In addition, programming at this low level of abstraction is cumbersome and error-prone. Recent trends in platform design exacerbate the problem: platforms are increasingly heterogeneous, e.g. including many general-purpose and specialized cores, parallel accelerators (GPUs), soft cores (FPGAs). As a consequence, even the development and tuning of applications for a specific machine configuration is complex and time consuming.

In sequential programming, the problem of having to recode for different machines was apparent three decades ago. The software engineering solution to this issue was to introduce levels of abstraction, effectively yielding a tree of refinements, from the problem specification to alternative target programs [1]. The derivation of a target program then follows a path down this tree. The transition from one node to the next can be described formally by a semantics-preserving program transformation or refinement. Conceptually, porting a program to a

B. Beckert et al. (Eds.): FMCO 2011, LNCS 7542, pp. 257–276, 2013.
© Springer-Verlag Berlin Heidelberg 2013

different machine configuration means backtracking to a previous node on the path and then following another path to a different target program.

This approach is not yet popular in the parallel programming setting. For example, typically parallel accelerators such as GPUs are programmed by directly leveraging on low-level accelerator-specific APIs (e.g. NVidia CUDA and OpenCL). Although these programming frameworks have been designed to keep narrow the gap between CPU and GPU programming style, there are still several differences, many of them emanating from the different nature of the architecture and even from the different models of computation of the GPUs. For example, when dealing with GPUs the programmer finds that all hardware facilities that are traditionally used to simplify the programming model have been removed (e.g. cache-coherence, branch prediction, virtual memory, global synchronizations) and so he/she must use very low-level mechanisms and must take into account a range of board specific information in order to obtain acceptable performance (e.g. local memory size, correct memory alignment, number of context, memory interleaving, etc.). Furthermore, the selection of which parts of an application should be executed on the GPU is completely the responsibility of the programmer and even if the code can be easily identified, there is no guarantee that it will be faster on the GPU than on a CPU. The programmer also has to manage data movement between the host processor's main shared memory and the GPU's core local memory taking care of memory alignment. Therefore porting code to GPUs, or developing from scratch an efficient code for GPUs, is not an easy task and can be a huge drain on resources. The typical code optimization curve grows very slowly and requires lots of performance testing and tuning, especially in industrial contexts where standard procedures for accurate testing and validation have to be performed.

Since the nineties, the "skeletons" research community [2] has been working on high-level languages and methods for parallel programming [3–6]. Skeleton programming requires the programmer to write a program using well-defined abstractions (called skeletons) derived from higher-order functions that can be parameterized to execute problem-specific code. Skeletons are parallel ab-initio and do not expose to the programmer the complexity of concurrent code, for example synchronization, mutual exclusion and communication. They instead specify abstractly common patterns of parallelism – typically in the form of parametric orchestration patterns – which can be used as program building blocks, and can be composed or nested like constructs of a programming language. A typical skeleton set includes the pipeline, the task farm, reduction and scan. For a given skeleton, usually, many efficient implementations for a given target platform may exist. Skeletons exhibit well-defined functional semantics, i.e. *what is computed*. As they describe parallelism exploitation paradigms, they also exhibit extra-functional behaviour, i.e. *how results are computed* [7], which can be also expressed by different realizations of the same pattern. For example, the functional composition operator $\circ$ can be interpreted as *pipeline* or as *sequence of functions*. We believe that the patterns/skeletons approach, which has been demonstrated to be effective for multi-core platforms (e.g. TBB [8] and Fastflow

[9] among others) can be used also with heterogeneous architectures to obtain a good trade-off between performance and code portability.

After incubation for over two decades in a quite restricted research community, skeletons gained renewed popularity with the arrival of multi-core platforms, the consequent diffusion of parallel programming frameworks, and their adoption in some successful programming frameworks, such as Intel Threading Building Block (TBB) [8]. Despite being complex to program, current multi-cores are almost uniform machines and in many cases they can be programmed with decent performance as if they were symmetric multiprocessors. However, this uniformity is progressively decaying with each new generation of machine: the current generation of multi-cores exhibit non-uniform memory access (typically cc-NUMA, i.e. cache-coherent Non Uniform Memory Access), while the next generation (e.g. IBM PowerEN, Intel MIC) will have specialized cores and accelerators to gain peak performance on critical tasks, and a non-uniform connection latency among cores and memory modules.

The heterogeneity and reduced connectivity of forthcoming platforms will make it all the more important for a programming framework to have the ability to generate parallel code according to different orchestration patterns and to map them on to different platforms in such a way that each task is run in the best suited executor and the synchronization and communication patterns are efficiently supported by the targeted platform. Currently, this activity is largely left to programmer expertise and is not effectively supported by development tools.

This work aims to make a step toward the formalization and the automation of this process. In particular, program refinement is proposed as an abstract tool to deal with the problem of the mapping of parallel activities onto heterogeneous cores (i.e. CPUs and GPUs). These parallel activities are assumed to be automatically generated by a high-level skeletal programming framework. In particular, skeletons are annotated with mapping information along a process of refinements. The first step annotates the tree with functional and extra-functional information such as data access, data dependencies, parallelism degree and so on; the next step maps the annotated tree on the given platform, taking into account the underlying target architecture; the last step executes the mapped tree and constantly notifies performance data to the upper levels so that, in case of performance degradation and driven by a suitable a performance model, the skeleton tree can be rewritten in a functional equivalent but better performing one. This is envisaged as the first step on the path to automated optimization of parallel codes onto heterogeneous platforms. In this respect, important milestones will include definition of a complete set of attributes to abstractly describe the key features of a specific parallel architecture, and the definition of suitable performance models able to drive the optimization process across the tree of refinements. These activities are currently ongoing in the ParaPhrase EC-STREP project.

The remainder of the paper is structured as follows: Sec. 2 introduces typical parallel programming patterns, while Sec. 3 introduces the abstract annotation

model which drives the mapping and rewriting of a user program refactored as a skeleton program. Sec. 4 provides some preliminary results of our approach implementation obtained on a heterogeneous platform. Finally, Sec. 5 discusses related work and Sec. 6 concludes the paper.

2 Parallelism Paradigms and Patterns

Attempts to reduce programming effort by raising the level of abstraction date back at least three decades. Notable results have been achieved by the *skeletal* approach [2, 10, 11], enabling *pattern-based* parallel programming. This approach appears to be becoming increasingly popular after reinforcement by several successful parallel programming frameworks [12–15].

Algorithmic skeletons capture common parallel programming paradigms (e.g. ForAll, MapReduce, Divide&Conquer, etc.) and make them available to the programmer as high-level programming constructs equipped with well-defined functional and extra-functional semantics [7]. Ideally, algorithmic skeletons address the difficult problems of parallel programming (i.e. concurrency exploitation, orchestration, mapping, tuning) moving them from the application design to development tools by capturing and abstracting the common paradigms of parallel programming and providing them with efficient implementations, i.e. a toolkit of code generation techniques and a pre-optimized run-time support.

Differences between algorithmic skeletons and parallel design patterns lie mainly in the motivations leading to these two apparently distinct concepts and in the research environments where they have been developed: the parallel programming community for algorithmic skeletons and the software engineering community for parallel design patterns. As far this work is concerned, the two concepts can be seen as synonymous.

Traditionally, in skeletal (and parallel pattern-based) programming the computation is organized according to application-independent high-level paradigms, which are usually categorized in three classes:

1. *Data Parallelism* is a method for parallelizing a single task by processing independent data elements in parallel. Data parallelism also supports loop-level parallelism where successive iterations of a loop working on independent or read-only data are parallelized in different flows-of-control and concurrently executed. *map* and *reduce* are instances of data parallelism.
2. *Task Parallelism* consists of running the same or different code on different executors (cores, machines, etc.). Task parallelism is usually explicit in the algorithm. Different flows-of-control (threads, processes, etc.) may communicate with one another as they work. Communication usually takes place to pass data from one thread/process to one or many others. The *farm* is a typical representation of such class of patterns
3. *Stream Parallelism* consists in the parallel processing of different items of a data stream, which can be either the input data or generated by the application's internal programming mechanisms (e.g. via asynchronous function

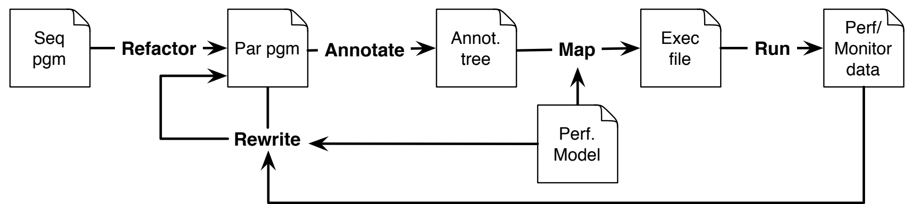


Fig. 1. Sketch of the proposed approach

calls). It can be used when there exists a partial or total order in a computation; the *pipeline* is a paradigmatic stream parallel pattern.

Pragmatically, a given computational problem typically admits several algorithmic solutions exploiting patterns in different classes, or different compositions of them. In addition, in many cases, patterns in different classes can simulate one another. The extent of this generality is dependent on the set of patterns provided by a specific framework, which can also be designed to target one or more application scenarios [15].

After Cole's seminal work [2], early proposals for skeletal programming frameworks have focused mainly on distributed memory platforms (e.g. clusters of workstations, grid); some of them, e.g. Google's MapReduce [13], have evolved in mainstream programming tools [16]. Recent proposals, following the platform architecture trend, have shifted the focus to include multi-cores and the shared address model; in addition to academic initiatives such as FastFlow (Sec. 4), it is worth mentioning consolidated industrial products such as the Intel Threading Building Block (TBB) library [8] and, to a limited extent, the Microsoft Task Parallel Library [17].

More recently, the skeletal approach has been proposed for GPGPUs and hybrid architectures: the SkePU framework is an example [18].

Some of these skeleton frameworks explicitly include stream parallelism as a major source of concurrency exploitation [7, 8, 12, 14]: rather than allowing programmers to connect stages into arbitrary graphs, basic forms of stream parallelism are provided to the programmer in high-level constructs such as *pipeline* (modelling computations in stages), *farm* (modelling parallel computation of independent data tasks), and *loop* (supporting generation of cycles in a stream graph and typically used in combination with a farm body to model Divide&Conquer computations).

3 A Refinement Process for Skeletons

3.1 Approach Overview

Our approach to parallelization via skeletons is depicted in Fig. 1. The starting point of the refinement process is a sequential program in which the user (or

eventually a tool) detects those parts of the code which can be parallelized. Parallelism can be introduced by a tool-assisted *Refactoring* process in which the user identifies patterns that can be captured by high level constructs (or calls to libraries) taking sequential code as parameter(s). This Refactoring process results in a high level program written as a composition of patterns/skeletons, i.e. a skeleton tree. The remainder of the development involves successively refining this skeleton through a series of stages to an implementation on a target architecture.

The *Annotate* phase uses a set of annotation rules to annotate the skeleton tree with an abstract description of the target architecture incorporating information such as number CPUs, number of GPUs, etc. In essence this annotated tree represents a set of possible mappings of tree to architecture.

The *Map* phase specializes the set of mappings implicit in the annotation tree to a particular mapping of components to resources. It uses more detailed target architecture specific detail (such as bandwidth of connections, speed of processors, etc.) and is informed by a performance model [19, 20] which allows qualitative assessment of alternative mapping strategies. The mapping phase produces an execution file that will be used by the architecture level for actively running the application (*Run* phase).

In addition to the above process of deriving an initial running program, one can envisage also a *Rewrite* phase which allows restructuring of the program as a result of feedback obtained from the running program. This Rewrite phase restructures the program in accordance with well-know functional equivalences between parallel patterns, again informed by the performance model. The result is a new (functionally equivalent) skeleton tree and so a new annotate phase can be commenced.

In the following section we will give a formal representation of the skeletons included in our semantic framework in order to define the *Annotate* and *Map* phases which are the focus of the current paper.

3.2 Skeleton Definition

Data and stream parallelism can be conveniently expressed using high-level patterns with well-defined functional semantics [7, 21, 22], whereas task parallelism, in the most general form, often subsumes low-level parallelism exploitation where synchronizations (as well as functional semantics) are deeply interwoven in the business code. For this reason, usually, they are not embedded in high-level pattern-based programming frameworks. In the following we use a generic two-tier pattern-based programming language including stream and data parallelism. Data parallel patterns can be nested within stream parallel patterns, but not vice-versa.

Let $\mathcal{P}$ be a pattern-based program, and $\mathcal{P}_{nc}$ a *non-cyclic* pattern-based program, i.e. a program not exhibiting cyclic data-dependencies among patterns. Let $\mathcal{P}_{sp}$ and $\mathcal{P}_{dp}$ be stream and data parallel high-level patterns, respectively which can be composed as follows:

$$
\begin{array}{ll}
\mathcal{P} & ::= \mathcal{P}_{nc} \mid parloop(\mathcal{P}_{nc}, E) \\
\mathcal{P}_{nc} & ::= \mathcal{P}_{sp} \mid \mathcal{P}_{dp} \\
\mathcal{P}_{sp} & ::= \mathcal{P}_{nc} \circ \mathcal{P}_{nc} \mid farm(\mathcal{P}) \\
\mathcal{P}_{dp} & ::= map(Seq) \mid reduce(Seq) \mid Seq \\
Seq & ::= \langle seq\ code \rangle \\
E & ::= \langle seq\ expression \rangle
\end{array}
$$

Here, for the sake of simplicity, the iterative usage of skeletons via the *loop* pattern, which can also be used to implement Divide&Conquer, is limited to the top level in order to simplify skeleton composition. Notice that in the most general case the *loop* pattern, if nested within other patterns, can receive data items from two different streams (*input* and *feedback* streams) and this requires proper management of non-determinism among them to avoid deadlock.

Patterns are assumed to exhibit a pure functional semantics, i.e. they can be defined as higher-order functions fully determined by their input-output behavior. As happens in the FastFlow framework [9], the approach can be extended to higher-order functions exhibiting a shared state. For example, using Ocaml-like notation to define the functional behavior, farm and pipeline skeletons can be described as follows:

```
let farm f x = (f x);;
let pipeline f g x = (g(f x));;
let map f x = Array.map f x;;
...
```

where streams, i.e. a (finite or infinite) sequence of values of the same type, are represented as lists. Patterns working on streams can be modelled accordingly, e.g.

```
let stream_parallel f x::y = (f x)::(stream_parallel f y);;
```

In $\mathcal{P}_{sp}$ the stream items are potentially computed in parallel. As an example, the farm skeleton uses a set of independent processing elements to compute the input tasks. Each time a new input task is available one of these resources is selected for the execution of the task, possibly using some kind of auto scheduling policy. The pipeline skeleton uses independent processing elements to compute the different stages in such a way that computation of stage i relative to task j can proceed concurrently (in parallel) with both the computation of stage $i-1$ for task $j+1$ and the computation of stage $i+1$ for task $j-1$.
On the other hand, in $\mathcal{P}_{dp}$ the parallel computation is applied to the input data as a whole. As an example, the map skeleton splits the input data collection into chunks on the basis of different policies and the same function is applied in parallel to each chunk by a different executor.

3.3 Skeleton Rewriting

As already mentioned, a skeleton is often defined by a functional semantics (*what is computed*) and a non-functional semantics (*how results are computed*) and it is

useful to make distinction, even informally, between them. Examples of a formal definition of (functional and non-functional) semantics for parallel patterns and streams can be found in [7, 22].

The functional semantics allows programmers to "compute" the function denoted by a pattern-based program. It also allows reasoning about program equivalence, in terms of the results computed, or to define semantics-preserving program transformations [21, 23]. These transformations can also be driven by some kind of analytical performance model associated with patterns, in such a way that only those rewritings leading to efficient implementations of the pattern are considered [21, 24, 25]. For instance, one can easily determine that the following two programs actually compute the same result, even if they exhibit different parallel behaviors:

```
let progA f g = stream_parallel (pipeline f g);;
let progB f g = stream_parallel (farm (pipeline f g));;
```

Also, streaming patterns can be *normalized* by reducing nesting of any depth of $farm$ and $\circ$ (i.e. *pipeline* in this context) to a $farm(pipeline()))$ [25].

Because patterns carry both a functional and non-functional semantics (thus the *intent* of the code [26]) they can also be used to support a generative approach to machine-specific run-time generation and optimization. For example, in the FastFlow framework (see Sec. 4), patterns are used to generate graphs of parallel activities and their orchestration in terms of (true) data dependencies.

We can refine this approach on a formal basis by defining a semantics allowing augmentation of the skeletal description provided by the application graph with mapping information and synchronization requirements with respect to the specific target architecture at hand. When the skeleton graph can be "rewritten" to a semantically equivalent one but enriched with information related to (potentially) optimal mapping, we can achieve better generation and optimization of the actual run-time to the specific machine at hand.

3.4 Annotation Semantics

Preliminary notation. For the sake of simplicity, we will provide an abstraction of a target architecture including one CPU (i.e a set of cores) and one or more GPUs, although our approach can be easily extended to a number of CPUs and GPUs available in a system. We will denote the set of $n \geq 0$ cores on the same CPU as

$$CPU = \{core_0, core_1, \ldots, core_n\}$$

representing the set of available cores on a given CPU.

$$GPU = \{gpu_0, gpu_1, \ldots, gpu_k\}$$

represents the set of available GPUs on a given architecture ($k \geq 0$).

Moreover, we assume that given a skeleton P, the mapping of P onto a given architecture x ($x \in CPU$ or $x \in GPU$) is represented by the notation P_x;

thus P_{core_1} will define the mapping of skeleton P locally onto $core_1$; P_{gpu_i} will define the mapping of P onto the *i-th* GPU available in the system; if P is a composite skeleton whose mapping could involve a set of computational resources $X = \{core_0, core_1\} \subseteq CPU$, then P_X will define the mapping between P and the sub-architecture represented by X.

Seq annotation. Our goal is to define an abstract semantics driving suitable mappings among (compositions of) skeletons as defined by Section 3 and the available abstract architectures at hand.
The base case is represented by the *Seq* skeleton, which will be simply mapped onto one of the cores available on the current CPU

$$\frac{x \in CPU = \{core_0, \dots, core_n\}}{Seq \rightarrow Seq_x} \tag{1}$$

or, since it could be encapsulated by a data parallel skeleton, it can be mapped onto a GPU

$$\frac{x \in GPU = \{gpu_0, \dots, gpu_n\}}{Seq \rightarrow Seq_x} \tag{2}$$

Farm annotation. Each instance of a farm will be rewritten in a notation highlighting the emitter (E) and the collector (C), in order to potentially allow different mappings of all the nodes composing such a skeleton. Thus, hold that

$$farm(P) = farm(E, P, C)$$

There are two possible configurations in mapping a farm: *i)* all the nodes are allocated on different cores of the same CPU:

$$\frac{E \rightarrow E_x \wedge P \rightarrow P_Y \wedge C \rightarrow C_z \wedge x, z \in CPU \wedge x \neq z \wedge Y \subseteq CPU - \{x, z\}}{farm(E, P, C) \rightarrow farm(E_x, P_Y, C_z)} \tag{3}$$

ii) emitter and collector are mapped onto different cores of the same CPU while the workers can be mapped onto a GPU

$$\frac{c_0, c_1 \in CPU \wedge X \subseteq GPU \wedge E \rightarrow E_{c_0} \wedge P \rightarrow P_X \wedge C \rightarrow C_{c_1}}{farm(E, P, C) \rightarrow farm(E_{c_0}, P_X, C_{c_1})} \tag{4}$$

With respect to such a rule we have to point out that in order to make suitable mappings, we should also take into account how the communication costs for moving data to and from the GPU influence the performance. In fact, placing the workers onto a GPU could be worthwhile if a huge set of tasks is ready to be delivered by the emitter for computation so that the workers can execute in a "dataparallel-like" mode on the set of input tasks; or, such mapping could be a good choice in those cases in which the task is, actually, a data parallel structure to be computed. Thus, while rule 4 is a good starting point for formalizing the mapping of workers onto a GPU, it needs to be further studied and enriched by data description details.

Parloop annotation. The parloop skeleton can be mapped to host the inner skeleton on any architecture while the condition is hosted on a CPU architecture:

$$\frac{x_2 \in CPU \wedge (X_1 \subseteq CPU \vee X_1 \subseteq GPU) \wedge P \rightarrow P_{X_1} \wedge E \rightarrow E_{X_2}}{parloop(P, E) \rightarrow parloop(P_{X_1}, E_{x_2})} \tag{5}$$

Since the evaluation of E defines whether the loop stops or continues to iterate, the rule above asserts that E is always evaluated on a CPU, while P (being a data parallel or a stream parallel skeleton) could be mapped onto a CPU or a GPU. Theoretically, if an iteration of P has been evaluated on x_i, the system memory of that node could still provide an up-to-date representation of data needed to proceed in the computation.

Map/Reduce annotation. Map and reduce can both be mapped on a CPU or a GPU architecture

$$\frac{x \subseteq GPU \vee x \subseteq CPU \wedge Seq \rightarrow Seq_x}{map(Seq) \rightarrow map(Seq)_x} \tag{6}$$

$$\frac{x \subseteq GPU \vee x \subseteq CPU \wedge Seq \rightarrow Seq_x}{reduce(Seq) \rightarrow reduce(Seq)_x} \tag{7}$$

Pipeline annotation. How a pipeline will be mapped depends at first on whether its stages are represented by stream parallel or data parallel skeletons, i.e. how data will flow through the graph and which dependencies among them are exploited. In the former case, the stages have to be placed on different cores (in order to exploit parallelism), but possibly of the same CPU (in order to minimize stream transfer costs):

$$\frac{x \neq y \wedge x, y \in CPU \wedge P' \rightarrow P'_x \wedge P'' \rightarrow P''_y \wedge P', P'' \in P_{sp}}{P' \circ P'' \rightarrow P'_x \circ P''_y} \tag{8}$$

In the latter case, a pipeline of data parallel skeletons can be mapped onto different cores (for instance, one stage per core) or onto different GPUs

$$\frac{x \neq y \wedge (x, y \in CPU \vee x, y \in GPU) \wedge P' \rightarrow P'_x \wedge P'' \rightarrow P''_y \wedge P', P'' \in P_{dp}}{P' \circ P'' \rightarrow P'_x \circ P''_y} \tag{9}$$

However, the pipeline of two data parallel stages could imply some synchronization steps between stages in the event of functional dependencies. For this reason, if the system provides just one GPU, the pure functional pipelining of two or more data parallel skeletons has to be rewritten in terms of a composition of stages (denoted by ";") because of the presence of some synchronization points that can serialize the execution.

$$\frac{P' \rightarrow P'_{gpu} \wedge P'' \rightarrow P''_{gpu} \wedge gpu \in GPU}{P' \circ P'' \rightarrow P'_{gpu}; P''_{gpu}} \tag{10}$$

Comp annotation. Comp is a skeleton (represented by ";" syntax) defining the sequential composition of two sub-skeletons which will be executed sequentially. This pattern is particularly useful in those rewritings in which part of a skeleton tree has to "collapse" into a sequential piece of code to provide improved performance, for example, in terms of communication costs.

$$\frac{x = y}{P'; P'' \rightarrow P'_x; P''_y} \tag{11}$$

The composed skeletons are mapped both on to the same target node.

3.5 Mapping Strategies

Data parallelism onto heterogeneous architectures. Let us assume we have the skeleton composition

$$map(Seq_1) \circ map(Seq_2)$$

Which suitable mappings can be provided, if the abstract target architecture is represented by the system $S = \{cpu_0, cpu_1, gpu_0\}$ where $CPU = \{cpu_0, cpu_1\}$ and $GPU = \{gpu_0\}$ and $\sharp GPU = 1$ represents the cardinality of the GPU set? As skeletons, Seq_1 and Seq_2 can be indifferently placed onto a CPU or a GPU architecture (rules 1 and 2), and so two branches are possible for the mapping of the two outer maps, since it holds that

$$\frac{Seq_1 \rightarrow (Seq_1)_x \wedge \forall x. x \in S}{map(Seq_1) \rightarrow map(Seq_1)_x}$$

and

$$\frac{Seq_2 \rightarrow (Seq_2)_x \wedge \forall x. x \in S}{map(Seq_2) \rightarrow map(Seq_2)_x}$$

However, at a higher level of the skeleton graph, the maps are composed by a *pipeline* operation. Recalling that our system provides just one GPU and two cores, we have two different options: *i)* we could place the pipeline on the same GPU but in a compositional manner so that they can eventually communicate via a shared memory system, i.e. by applying rule 10

$$\frac{map(Seq_1) \rightarrow map(Seq_1)_{gpu} \wedge map(Seq_2) \rightarrow map(Seq_2)_{gpu}}{map(Seq_1) \circ map(Seq_2) \rightarrow map(Seq_1)_{gpu}; map(Seq_2)_{gpu}}$$

ii) we could place the pipeline so that the first map is executed on cpu_1 and the second one on cpu_2 and the stages will then communicate via a stream of data, thus applying rule 9.

$$\frac{cpu_0, cpu_1 \in CPU \wedge map(Seq_1) \rightarrow map(Seq_1)_{cpu_0} \wedge map(Seq_2) \rightarrow map(Seq_2)_{cpu_1}}{map(Seq_1) \circ map(Seq_2) \rightarrow map(Seq_1)_{cpu_0} \circ map(Seq_2)_{cpu_1}}$$

Which of these two options will be actually chosen will depend on the ability of the system to make predictions on the cost of each configuration. Providing the semantics with a cost model allowing an estimation of each candidate configuration will be the goal of future work.

noise	Seq (1 CPU)	8 cores + 24 CPUs	8 cores + 1 GPUs
10 %	32.0 s	1.8 s	1.9 s
50 %	162.1 s	6.5 s	2.3 s
90 %	290.0 s	10.9 s	2.8 s

Lena - 30% noise
Lena - 50% noise
Lena - 90% noise
Lena 30% - Restored PSNR=35.1 MAE=1.2
Lena 50% - Restored PSNR=31.9 MAE=2.3
Lena 90% - Restored PSNR=22.5 MAE=11.3

Fig. 2. Left) Execution time of different configuration of the *Detect+Restore* functions on Lena image. Right) Restoration result with PSNR (Peak Signal-to-Noise Ratio) and MAE (Mean Absolute Error).

Bringing down data transfer costs. Let now assume that we have the following skeleton composition

$$Seq1 \circ Seq2 \circ Seq3$$

From a functional perspective the *pipeline* operation exploits the associative properties so that

$$(Seq1 \circ Seq2) \circ Seq3 \equiv Seq1 \circ (Seq2 \circ Seq3)$$

However, from a mapping point of view, these two options could imply very different performance effects: for example, let us suppose that a number of dual-core CPUs are available so that $CPU_0 = \{core_0, core_1\}$ and $CPU_1 = \{core_2, core_3\}$: the better mappings are those assigning cores belonging to the same CPU to (possibly) contiguous stages. Thus, while

$$(Seq1_{core_0} \circ Seq2_{core_1}) \circ Seq3_{core_2}$$

$$(Seq1_{core_1} \circ Seq2_{core_0}) \circ Seq3_{core_2}$$

$$Seq1_{core_1} \circ (Seq2_{core_2} \circ Seq3_{core_3})$$

$$Seq1_{core_0} \circ (Seq2_{core_2} \circ Seq3_{core_3})$$

would be good combinations since they minimize the extra-CPU communication to just one occurrence, all the other combinations, such as for example

$$Seq1_{core_0} \circ (Seq2_{core_2} \circ Seq3_{core_1})$$

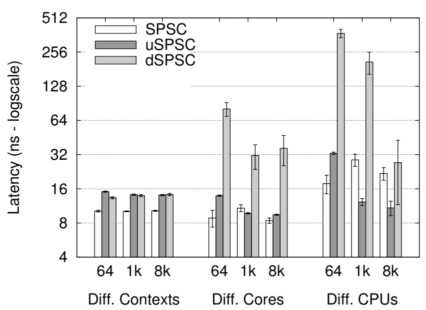
will need two extra-CPU communications, maybe accessing a shared memory or even across the network in the case of a cluster. In Sec. 4 we will see a concrete instantiation of this principle applied to a specific architecture.

4 Preliminary Results

In the current section we will exemplify the proposed methodology through some examples implemented on top of FastFlow, a parallel programming framework aimed at simplifying the development of applications for multi-core platforms, whether these applications are brand new or ports of existing legacy codes [27]. FastFlow promotes pattern-based programming and has been specifically designed to efficiently support fine-grained parallel computations. The FastFlow patterns can be arbitrarily nested to model increasingly complex parallelism exploitation patterns. The FastFlow implementation guarantees an efficient execution of the skeletons on currently available multi-core systems by building the skeletons themselves on top of a library of lock-free producer/consumer queues. The workstation on which we performed the tests is a "homogeneous" Intel Nehalem microarchitecture equipped with 4 eight-core double context Xeon E7-4820 @2.0GHz with 18MB L3 shared cache, 256K L2, and 24 GBytes of main memory with Linux x86_64.

Current multi-core machines, such as Intel or AMD multi-core platforms are typically programmed and managed as if they were symmetric multiprocessors. However, the relation between performance and mapping of parallel activities onto core can be easily shown. For example, Fig. 3 reports the latency of three different implementations of the FastFlow Single-Producer Single-Consumer queue on the tested platform: a bounded array-based queue (SPSC), a dynamically linked-list queue (dSPSC) and an unbounded array-based queue (uSPSC). All implementations are lock-free and particularly optimized to avoid cache invalidations [28].The queue implementations are compared on three different mappings for the producer (P) and the consumer (C): 1) P and C are placed on two different hardware contexts of the same core; 2) P and C are placed on two different cores of the same socket; 3) P and C are placed on two different sockets. As can be seen, the dSPSC queue is particularly sensitive to mapping as the latency from one mapping to another changes the performance more than two orders of magnitude. This gap is expected to grow in forthcoming platforms with increasing core count and platform heterogeneity.

In the Table of Fig. 3, we report the performance obtained when running a very simple benchmark test where one 3 stage pipeline computes a stream of 1M tasks (double elements). Each stage is connected with the previous and following one (if present) using the FastFlow dSPSC unbounded queue. The first stage mainly generates the stream of tasks whereas the other two stages apply on each input a function computing a trigonometric computation. The third stage of the pipeline (s3) is the most computationally demanding. In this test we consider 4 possible mapping strategies for the three stages on the considered architecture. The best performance is obtained when the first 2 stages are mapped on the same core (different contexts) of the first CPU and the third stage is mapped on the second CPU (mappingC in the Table). In this way we are able to obtain a good trade-off between communication costs and computation. In fact, the first and the second stage do not interfere too much when placed on the same context since the first stage does not perform any significant numerical computation; instead,



mapping strategies	Compl. Time (ms)
mappingA	360
mappingB	530
mappingC	295
mappingD	480

– mappingA) s1,s2,s3 on adjacent cores of CPU1;
– mappingB) s1 on CPU1, s2 on CPU2 and s3 on CPU3;
– mappingC) s1 and s2 on the same context of one core of CPU1 and s3 on CPU2;
– mappingD) s2 and s3 on the same context of one core of CPU1 and s1 on CPU2.

Fig. 3. Left) Latency of 3 different implementations of FastFlow queues tested with three different mapping for the Producer and the Consumer threads [27]. Right) Performance obtained for the 3-stage pipeline(s1,s2,s3) benchmark varying the stage mapping.

they are able to benefit from the lower level cache to increase communication performance.

In order to validate the proposed methodology we describe a prototypical example, an image restoration application. The edge-preserving denoiser is a two-step filter for removing salt-and-pepper noise (see Fig. 2). In the first step, an adaptive median filter is used to identify the set of noisy pixels; in the second step, these pixels are restored according to an iterative variational approach up to convergence. The detailed description of the sequential algorithm is beyond the scope of this paper; it ensures state-of-the-art restoration quality and execution time, and is able to restore also very noisy images (e.g. up to 90% random noisy pixels) [29]. The same algorithm can also be used to restore video streams by iterating frame-by-frame the detect-denoise filters.

Pattern/Skeleton selection. Let $ReadImg, Detect, Restore, WriteImg, Fixpoint$ be chunks of sequential code (e.g. functions, i.e. *Seq*). The core of the edge-preserving denoiser can be sketched as

```
Img=ReadImg;
NoisySet=Detect(Img);
while(!Fixpoint(MAE((Img)){Img=Restore(Img,NoisySet);}
WriteImg(Img);
```

which can be iterated in a loop to realize a video version that simply repeats the same process on successive video frames. Notice that the *Restore* process is iterated up to *fixpoint* times by way of the *Fixpoint* function. The *fixpoint* is reached when the restoration process brings no improvement in the "quality" of the image across two successive iteration. The quality of the image is usually measured in term of *Peak Signal-to-Noise Ratio* (PSNR) or *Mean Absolute Error*

(MAE). The video version can be sketched as follows:

```
while(true){
  Img=ReadImg;
  NoisySet=Detect(Img);
  while(!Fixpoint(MAE(Img))){Img=Restore(Img,NoisySet);}
  WriteImg(Img);
}
```

where the two filters *Detect* and *Denoise* are both executed sequentially and successively. Again, the latter filter is iterated up to *fixpoint* times. In order to detect when the *fixpoint* value is reached, the MAE filter has to be computed at each iteration. Computing MAE requires the analysis of the whole image `Img`. The visual effect on a noisy image of the two filters is shown in Fig. 2 right), together with the quality measures obtained (PSNR and MAE). The two principal filters can be parallelized in a data-parallel fashion using the *map* pattern, as follows:

```
while(true){
  Img=ReadImg;
  NoisySet=map(Detect(Img));
  while(!Fixpoint(MAE(Img))){Img=map(Restore(Img,NoisySet));}
  WriteImg(Img);
}
```

The MAE computation can be also parallelized in a data-parallel fashion using the *reduce* pattern. In addition, the parallelized versions of *Restore* and MAE can be composed and executed in a parallel loop in such a way that the whole restoration loop can be wrapped and possibly offloaded to an accelerator.

```
while(true){
  Img=ReadImg;
  NoisySet=map(Detect(Img));
  parloop((Fix=reduce(MAE,Img)) ∘ (Img=map(Restore(Img,NoisySet))),!Fix);
  WriteImg(Img);
}
```

Annotate. From a semantic perspective and by following the syntax presented in this proposal, the preceding piece of code can be represented as

$$map(Seq_D) \circ parloop((reduce(Seq_M) \circ map(Seq_R)), E)$$

where we assume $Seq(Detect(Img)) = Seq_D$, $Seq(Restore(Img, NoisySet)) = Seq_R$, $Seq(MAE) = Seq_M$ and $E = !Fix$; our set of rules is then able to derive for us the annotation of the syntax tree associated to this composite skeleton as follows:

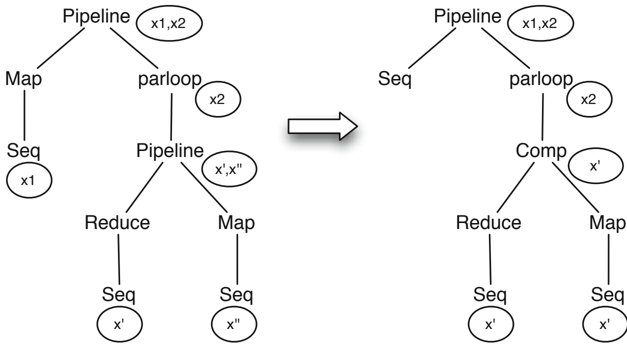


Fig. 4. The annotated skeleton tree of the application: its implementation and performance are parametric w.r.t. x_1, x_2, x', x''

$$map(Seq_D) \circ parloop((reduce(Seq_M) \circ map(Seq_R)), E)$$
$$\rightarrow \{\text{let } x_1, x_2 \text{ two cores, rule 9 holds and } Seq_D \rightarrow Seq_{Dx_1} \text{ hold }\}$$

$$map(Seq_{Dx_1}) \circ parloop((reduce(Seq_M) \circ map(Seq_R)), E)_{x_2}$$

$$\rightarrow \{\text{ rule 5 and } x', x'' \text{ potentially fresh id}\}$$

$$map(Seq_{Dx_1}) \circ parloop((reduce(Seq_M)_{x'} \circ map(Seq_{Rx''})), E_{x_2})_{x_2}$$

The annotated tree associated with such mapping evaluation is depicted in Fig. 4 (left) where x_1, x_2, x', x'' could identify a set of different or overlapping cores. In addition, each of x_1, x_2, x', x'' can also be either CPU or GPU cores. The performance model provides the information needed to choose the mapping which gives the best performance.

It is worth pointing out that at this level the semantic framework could also be able to define (under specific performance requirements) an alternative skeleton tree, functional equivalent to the preceding one for which a new set of mapping alternatives could hold. Fig. 4 suggests a possible rewriting of the skeleton in which $map(Seq_{D_1})$ has been rewritten as sequential and the pipeline iterated by *parloop* has collapsed in a *comp*. Such skeleton could be eligible if, for instance, the costs involved in the implementation of the map and the pipeline are too high with respect to a sequential execution, possibly because of a too fine grain computation.

Mapping. According to the methodology introduced in Sec. 3, the parallelized versions of *Detect* and *Restore* can be mapped onto different processors (i.e. $x_1 \neq x''$), resulting in different performance figures. For example, Fig. 2 presents, together with the sequential execution time, the performance when the *Detect* and *Restore* filters are executed in parallel using, respectively, 8 and 24 cores of the 32 cores of the Intel workstation described at the beginning of this section. Alternatively, the restoration loop can be offloaded to a GPGPU (NVidia

Tesla 448 cores) with even better results. In this mapping process, the described methodology is intended to ensure that only appropriate compositions of patterns are mapped onto the GPGPU.

5 Related Work

Early proposals of pattern-based parallel programming frameworks have been focused mainly on distributed memory platforms, such as clusters of workstations and grids [12, 30]. Google MapReduce [13] brings to the mainstream of out-of-core data processing the map-reduce paradigm. All these skeleton frameworks provide several parallel patterns (algorithmic skeletons) covering mostly task and data parallelism. These patterns can usually nested to model more complex parallelism exploitation patterns according to the constraints imposed by the specific programming framework. More recent pattern-based frameworks, following the platform architecture trend, have shifted the focus to multi-cores and the shared address model; in addition to FastFlow, it is worth mentioning the Intel Threading Building Block (TBB) library [8], and to a limited extent the Microsoft Task Parallel Library [17]. All of them are certainly higher-level compared to the Pthread library that has been used in the shared memory implementations of classification algorithms previously mentioned. The main features of these and other frameworks are surveyed in [11].

Programming frameworks based on algorithmic skeletons have been recently introduced to alleviate the task of the application programmer when targeting data parallel computations on GPUs. In Muesli [31] the programmer must explicitly indicate whether GPUs are to be used for data parallel skeletons. SkePU [18] provides programmers with GPU implementations of map and reduce skeletons and relies on StarPU for the execution of stream parallel skeletons (pipe and farm).

In addition to pattern-based frameworks, other high-level programming frameworks also aim to simplify the design of efficient applications on multi-cores and thus are related to FastFlow and to the present work. StreamIt [14] is an explicitly parallel programming language based on the Synchronous Data Flow model that enables the assembly of program modules (called filters) in a *pipeline* fashion, possibly with a *FeedbackLoop*, or according to a *SplitJoin* data-parallel schema. Streaming applications are also targeted by TBB through the *pipeline* construct, which also provides programmers with thread-safe containers and some parallel patterns (called "algorithms"); TBB does not support any kind of non-linear streaming network, which thus has to be embedded in a pipeline with significant programming and performance drawbacks. Intel's Concurrent Collections (CnC), which declaratively models concurrent activities as data streams and control dependencies, has been recently proposed as a candidate substrate for parallel patterns [32]. *OpenMP* [33] is a popular thread-based framework for multi-core architectures mostly targeting data parallel programming even if it is currently being extended to incorporate stream processing. OpenMP supports, by way of language pragmas, the low-effort parallelization

of sequential programs; however, these pragmas are mainly designed to exploit loop-level data parallelism (e.g. *do_independent*). CnC and OpenMP do not natively support either *farm* or *Divide&Conquer* patterns, even though they can be simulated with lower-level features.

MCUDA [34] is a framework to mix CPU and GPU programming. In MCUDA it is mandatory to define kernels for all available devices but the framework can not make any assumptions about the relative performance of the supported devices.

Recently OpenACC [35] has been proposed by some major vendors as a possible new standard for programming GPUs and HW accelerators in general. Like OpenMP it is based on a set of pragma directives allowing automatic acceleration of loops and parallel regions by directly offloading computation on the accelerator. Our approach differs from #pragma-based approaches because we require an explicit parallelization of the code thus making it possible to avoid cluttering the sequential code with complex directives. Furthermore, in our vision, the declarative approach gives, in many cases, less control to the programmer and hence lessens the possibility to exploit all the available parallelism in the application.

Overall, our approach aims to provide a high-level programming model based on algorithmic skeletons and a high-level skeleton-based intermediate representation with mapping annotations, which are used for taking mapping decisions. In our vision such an approach is able to ensure portability of the parallel code onto different heterogeneous platforms while maintaining good performance.

6 Conclusions

The mapping problem, and in general the optimization of parallel code for current and next generation parallel platforms is a particularly important problem as it might significantly affect performance and efficiency of applications. Ideally, solutions to this problem should address performance portability while not requiring excessive effort on the part of the application developer. In this respect, the pattern-based approach has been demonstrated to have the potential to address the problem. In this position paper we have stated the problem and the approach we are undertaking to define an intermediate formalism to support the compilation and the optimization of patterns on heterogeneous multi-core and many-core platforms. The intermediate language basically aims to equip patterns with several platform attributes in such a way that suitable mapping and scheduling heuristics aimed at generating optimized run-time code can be derived.

Acknowledgment. The work described in this paper is supported by the EU ParaPhrase project (`http://www.paraphrase-ict.eu`, 2011-2014).

References

1. Parnas, D.L.: On the design and development of program families. IEEE Trans. on Software Engineering SE-2(1), 1–9 (1976)
2. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computations. Research Monographs in Par. and Distrib. Computing. Pitman (1989)

3. Botorog, G.H., Kuchen, H.: Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In: Proc. of the 5th International Symposium on High Performance Distributed Computing, HPDC 1996, pp. 243–252. IEEE Computer Society Press (1996)
4. Darlington, J., Guo, Y., Jing, Y., To, H.W.: Skeletons for structured parallel composition. In: Proc. of the 15th Symposium on Principles and Practice of Parallel Programming (1995)
5. Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., Vanneschi, M.: P^3L: A Structured High level programming language and its structured support. Concurrency Practice and Experience 7(3), 225–255 (1995)
6. Hamdan, M., King, P., Michaelson, G.: A scheme for nesting algorithmic skeletons. In: Hammond, K., Davie, T., Clack, C. (eds.) Proc. of the 10th International Workshop on the Implementation of Functional Languages, IFL 1998, Department of Computer Science, University College London, pp. 195–211 (1998)
7. Aldinucci, M., Danelutto, M.: Skeleton based parallel programming: functional and parallel semantics in a single shot. Computer Languages, Systems and Structures 33(3-4), 179–192 (2007)
8. Intel Corp.: Threading Building Blocks (2011)
9. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multi-core. In: Pllana, S., Xhafa, F. (eds.) Programming Multi-core and Many-core Computing Systems. Parallel and Distributed Computing. Wiley (2012)
10. Cole, M.: Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. Parallel Computing 30(3), 389–406 (2004)
11. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. Software: Practice and Experience 40(12), 1135–1160 (2010)
12. Vanneschi, M.: The programming model of ASSIST, an environment for parallel and distributed portable applications. Parallel Computing 28(12), 1709–1732 (2002)
13. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: Usenix OSDI 2004, pp. 137–150 (December 2004)
14. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A Language for Streaming Applications. In: CC 2002. LNCS, vol. 2304, pp. 179–196. Springer, Heidelberg (2002)
15. Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K.: A view of the parallel computing landscape. Comm. of the ACM 52(10), 56–67 (2009)
16. Apache Software Foundation: Hadoop (2008), `http://hadoop.apache.org/`
17. Leijen, D., Hall, J.: Optimize managed code for multi-core machines. MSDN Magazine (October 2007)
18. Enmyren, J., Kessler, C.W.: Skepu: a multi-backend skeleton programming library for multi-gpu systems. In: Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications, HLPP 2010, pp. 5–14. ACM, New York (2010)
19. Aldinucci, M., Coppola, M., Danelutto, M.: Rewriting skeleton programs: How to evaluate the data-parallel stream-parallel tradeoff. In: Gorlatch, S. (ed.) Proc. of CMPP: Intl. Workshop on Constructive Methods for Parallel Programming, Fakultät für mathematik und informatik, Uni. Passau, Germany, pp. 44–58 (May 1998)

20. Skillicorn, D.B., Cai, W.: A cost calculus for parallel functional programming. J. Parallel Distrib. Comput. 28(1), 65–83 (1995)
21. Aldinucci, M., Gorlatch, S., Lengauer, C., Pelagatti, S.: Towards parallel programming by transformation: The FAN skeleton framework. Parallel Algorithms and Applications 16(2-3), 87–121 (2001)
22. Caromel, D., Henrio, L., Leyton, M.: Type safe algorithmic skeletons. In: 16th Euromicro Intl. Conference on Parallel, Distributed and Network-Based Processing, PDP, Toulouse, France, pp. 45–53. IEEE (February 2008)
23. Gorlatch, S., Lengauer, C., Wedler, C.: Optimization rules for programming with collective operations. In: Proc. of the 13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing, IPPS/SPDP 1999, pp. 492–499. IEEE Computer Society Press (1999)
24. Skillicorn, D.B., Cai, W.: A cost calculus for parallel functional programming. Journal of Parallel and Distributed Computing 28, 65–83 (1995)
25. Aldinucci, M., Danelutto, M.: Stream parallel skeleton optimization. In: Proc. of PDCS: Intl. Conference on Parallel and Distributed Computing and Systems, Cambridge, Massachusetts, USA, pp. 955–962. IASTED, ACTA Press (November 1999)
26. Pottenger, B., Eigenmann, R.: Idiom recognition in the Polaris parallelizing compiler. In: Proc. of the 9th Intl. Conference on Supercomputing, ICS 1995, pp. 444–448. ACM Press, New York (1995)
27. Aldinucci, M., Torquati, M.: FastFlow website (2009), http://mc-fastflow.sourceforge.net/
28. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: An Efficient Unbounded Lock-Free Queue for Multi-core Systems. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 662–673. Springer, Heidelberg (2012)
29. Aldinucci, M., Drocco, M., Giordano, D., Spampinato, C., Torquati, M.: A parallel edge preserving algorithm for salt and pepper image denoising. Technical Report 138/2011, Università degli Studi di Torino, Dip. di Informatica, Italy (May 2011)
30. Kuchen, H.: A Skeleton Library. In: Monien, B., Feldmann, R. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 620–629. Springer, Heidelberg (2002)
31. Ernsting, S., Kuchen, H.: Data parallel skeletons for gpu clusters and multi-gpu systems. In: Proceedings of PARCO 2011. IOS Press (2011)
32. Newton, R., Schlimbach, F., Hampton, M., Knobe, K.: Capturing and composing parallel patterns with Intel CnC. In: Proc. of USENIX Workshop on Hot Topics in Parallelism, HotPar 2010, Berkley, CA, USA (June 2010)
33. Park, I., Voss, M.J., Kim, S.W., Eigenmann, R.: Parallel programming environment for OpenMP. Scientific Programming 9, 143–161 (2001)
34. Stratton, J.A., Stone, S.S., Hwu, W.-M.W.: MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 16–30. Springer, Heidelberg (2008)
35. Khronos Compute Working Group: OpenACC Directives for Accelerators (November 2012), http://www.openacc-standard.org

PRO3D, Programming for Future 3D Manycore Architectures: Project's Interim Status

Christian Fabre[1,7], Iuliana Bacivarov[3], Ananda Basu[2,7], Martino Ruggiero[4], David Atienza[6], Éric Flamand[5], Jean-Pierre Krimm[1,7], Julien Mottin[1,7], Lars Schor[3], Pratyush Kumar[3], Hoeseok Yang[3], Devesh B. Chokshi[3], Lothar Thiele[3], Saddek Bensalem[2,7], Marius Bozga[2,7], Luca Benini[4], Mohamed M. Sabry[6], Yusuf Leblebici[6], Giovanni De Micheli[6], and Diego Melpignano[5]

[1] CEA, LETI, Campus Minatec, Grenoble, France
{christian.fabre1,jean-pierre.krimm,julien.mottin}@cea.fr
[2] VERIMAG, Centre Équation, 2 av. de vignate, 38610 Gières, France
{ananda.basu,saddek.bensalem,marius.bozga}@imag.fr
[3] ETHZ, Computer Engineering and Networks Laboratory, 8092 Zürich, Switzerland
{bacivarov,lschor,kumarpr,hyang,dchokshi,thiele}@tik.ee.ethz.ch
[4] Università di Bologna, Bologna, Italy
{martino.ruggiero,luca.benini}@unibo.it
[5] STMicroelectronics, Grenoble, France
{eric.flamand,diego.melpignano}@st.com
[6] EPFL, Lausanne, Switzerland
{david.atienza,mohamed.sabry,yusuf.leblebici,giovanni.demicheli}@epfl.ch
[7] CRI – Centre de recherche intégrative, 7 al. de palestine, 38610 Gières, France
http://www.cri-grenoble.fr

Abstract. PRO3D tackles two important 3D technologies, that are Through Silicon Via (TSV) and liquid cooling, and investigates their consequences on stacked architectures and entire software development. In particular, memory hierarchies are being revisited and the thermal impact of software on the 3D stack is explored. As a key result, a software design flow based on the rigorous assembly of software components and monitoring of the thermal integrity of the 3D stack has been developed. After 30 months of research, PRO3D proposes a complete tool-chain for 3D manycore, that integrates state-of-the-art tools ranging from system-level formal specification and 3D exploration, to actual programming and runtime control on the 3D system. Current efforts are directed towards extensive experiments on an industrial embedded manycore platform.

1 Introduction

Three dimensional stacked integrated circuits (3D ICs) are extremely attractive for overcoming the barriers in interconnect scaling, offering an opportunity to continue CMOS performance trends for the next decade. With the ever increasing demand for higher data rates and performance as well as multi-functional capabilities in circuits, vertical integration of IC dies using through-silicon vias

B. Beckert et al. (Eds.): FMCO 2011, LNCS 7542, pp. 277–293, 2013.
© Springer-Verlag Berlin Heidelberg 2013

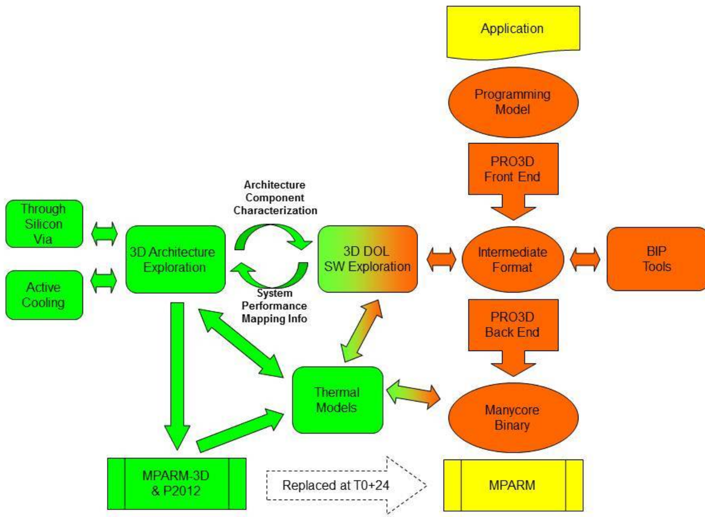


Fig. 1. PRO3D Exploration & Design Flow for 3D Manycore

is envisioned to be one of the most viable solutions for the development of new generation of electronic products. 3D integration of multi-core processors offers massive bandwidth improvements while reducing the effective chip footprint. However 3D integration introduces several challenges, mostly related to the following factors:

– increasing amount of logic that can be placed onto a single 3D IC,
– related thermal dissipation problem,
– a necessary shift in programming models towards more parallelism.

The manycore revolution and the ever-increasing complexity of 3D ICs is dramatically changing system design, analysis and programming of computing platforms. Future 3D architectures will feature hundreds of cores and on-chip memories connected through complex 3D communication architectures. Moreover, the third dimension leads to a tremendous increase in heat dissipation per unit area of the chip. This in turn results in higher chip temperatures and thermal stress, hence, (a) limiting the performance and reliability of the chip and (b) requiring software development tools and runtime to address thermal concerns.

PRO3D addresses the above mentioned challenges and proposes Fig. 1 a software and exploration design flow based on the rigorous assembly of software components and monitoring of the thermal integrity of the 3D stack: Section 2 investigates memory hierarchies and thermal-aware architectural exploration (corresponding to *TSV, Active Cooling, MPARM, MPARM-3D* & *Architecture Exploration* in Fig. 1 above); Section 3 details the active cooling strategy for the proposed 3D stacks (correspponding to *TSV, Active Cooling* & *Thermal Models*); Section 4 investigates system-level formal solutions that guarantee thermal properties during mapping of application tasks on the 3D architecture (corresponding

to *3D DOL SW Exploration & Thermal Models*); Section 5 describes formal verification methods for PRO3D systems (From *Programming Model* to *Manycore Binary*, plus *3D DOL & BIP Tools* in Fig. 1); Section 6 provides an overview of STHORM, the PRO3D target platform (corresponding to *STHORM*). Finally, our current achievements are summarized in Section 7.

2 3D Architectural Exploration

With three dimensional stacked integrated circuits (3D ICs), accurate 3D thermal-aware system-level architectural exploration plays a fundamental role in system design. System-level architectural explorations and thermal issues have so far been addressed independently at different levels of the system design. Hence, new methodologies that address the heat removal problem concurrently at all stages and levels of the 3D chip design need to be developed and to be exploited by high-level software programming frameworks. Designers of upcoming 3D chip will need new distinctive tools for thermal-aware 3D architectural exploration, enabling a cooling-aware design of 3D ICs.

PRO3D has developed a flexible virtual platform infrastructure (VPI) for modelling and analysis of 3D integrated architectures and memory systems, as well as accurate thermal models for calculating the costs of operating the cooling, determining the overall energy budget and performing run-time thermal management.

MPARM [28] has been used as main VPI tool for design space explorations. It is a virtual SoC platform based on the SystemC simulation kernel, which could be used to model both HW & SW of complex systems. The system architecture simulated by the default MPARM distribution is represented by a homogeneous multicore system based on shared bus communication. During PRO3D, MPARM has been enhanced with several HW parametric models of the main micro-architectural components of a 3D integrated interconnect and memory hierarchy [6,31], and a support of modular plug-ins for thermal models interfacing. The new models are highly parametric, flexible and customizable.

2.1 Functional Modelling of 3D Memory Hierarchy

To keep the pace of Moore's law, future 3D-IC platforms will be embracing the many-core paradigm, where a large number of simple cores are integrated onto the same die. Current examples of many-cores include GP-GPUs such as NVIDIA *Fermi* [23], the *HyperCore Architecture Line (HAL)* [24] processors from Plurality, or ST Microelectronics *Platform 2012* [5,20,36].

While there is renewed interest in Single Instruction Multiple Data (SIMD) computing, thanks to the success of GP-GPU architectures, strict instruction scheduling policies enforced in current GP-GPUs are being relaxed in the most recent many-core designs to exploit data parallelism in a flexible way. Single Program Multiple Data (SPMD) parallelism can thus be efficiently implemented in these designs, where processors are not bound to execute the same instruction stream in parallel to achieve peak performance.

All of the cited architectures share a few common traits: their fundamental computing tile is a tightly coupled cluster with a shared multibanked L1 memory for fast data access and a fairly large number of simple cores, with ≈ 1 Instruction Per Cycle (IPC) per core. Key to providing I-fetch bandwidth for cluster-based CMP is an effective instruction cache architecture design, therefore a detailed design space exploration and analysis have been conducted to evaluate how microarchitectural differences in L1 instruction cache architectures may affect the overall system behavior and IPC.

We analyzed and compared the two most promising architectures for instruction caching targeting tightly coupled CMP clusters, namely private instruction caches per core and shared instruction cache per cluster.

Experimental results showed that private cache performance can be significantly affected by the higher miss cost; on the other hand the shared cache has better performance, with speedup up to almost 60%. However, it is very sensitive to execution misalignment, which can lead to cache access conflicts and high hit cost [6].

2.2 Enabling Thermal-Aware System-Level Architectural Exploration

PRO3D has also produced 3D-ICE, a compact transient thermal model (CTTM) for liquid cooling that provides fast and accurate thermal simulations of 3D ICs with inter-tier microchannel cooling [42]. 3D-ICE can accurately predict the temporal evolution of chip temperatures when system parameters (heat dissipation, coolant flow rate, etc.) change during dynamic thermal management. We have validated the accuracy of the model with a commercial computational fluid dynamics simulation tool as well as measurement results from a 3D test IC and have foudn a maximum error of 3.4 % in temperature.

PRO3D has also defined and characterized (electrically and thermally) a 3D integration process flow [21,35,45] that combines TSV and microchannels fabrication for liquid cooling of multiple tiers and has developed 3D-ICE [34], a complete transient thermal simulation tool that can be used to validate 3D integration stacks of multi-core designs in a very early stage of the design flow, thus enabling much more thermally-balanced and controlled 3D multi-core designs. These high-level technology models of complete 3D stacks have been successfully used to validate the effects of the cooling methods while executing benchmarks in the VPI [18].

3 Thermal Management

Inter-tier liquid cooling is a recently proposed and a promising thermal packaging solution to counter the aggravated thermal issues arising from vertical stacking in 3D-multiprocessor ICs [8]. With this packaging solution, inter-tier thermal resistances are reduced considerably, enabling the 3D ICs to operate at much lower temperatures than those with conventional heat sinks [30,26].

However, inter-tier liquid cooling also brings with it new design-time and run-time challenges for the designers. For instance, a serious challenge that single-phase liquid cooling brings is the increased thermal gradient. The sensible heat absorption that occurs as the coolant flows along the microchannels raises its temperature [34]. This results in an increase of coolant temperature from inlet to the outlet, which in turn, results in an undesirably augmented thermal gradient on the IC surface [30]. These gradients cause uneven thermally-induced stresses on different parts of the IC, significantly undermining overall system reliability [10].

In this respect, we propose a novel design-time thermal balancing technique by modulating the microchannel width from inlet to outlet, without adding to the existing fabrication costs. This technique, referred to as *channel modulation*, relies on the well-known observation that the thermal resistance of microchannel heat sinks reduces with increasing aspect ratio of the channel cross-section [41]. Our proposed work provides an optimal solution for thermal balancing and hotspot minimization. This work contributes to providing an additional dimension of design-space exploration, in the form of channel modulation, to IC designers for the purpose of thermal balancing.

3.1 Thermal Model and Problem Formulation

The goal of our optimization is to find a sequence of channel widths, as a function of the distance from the inlet, which minimizes the intended cost function: the temperature gradient. Hence, the **steady-state temperatures** of the 3D IC must be written as a function of this distance in the analytical formulation, with the channel widths as an input parameter. In other words, if the distance from the inlet is measured along the coordinate axis z, then we need to find an equation of the form:

$$\frac{d}{dz}\mathbf{T}(z) = \Phi(z, \mathbf{w_C}(z), \mathbf{T}(z)), \tag{1}$$

where $\mathbf{T}(z)$ is the steady-state temperatures vector on the IC and $\mathbf{w_C}(z)$ is a vector of width functions of different microchannels written as a function of z. Our goal, then, is to find $\mathbf{w_C}(z)$ that minimizes the gradients in $\mathbf{T}(z)$. It is important to mention that there are five heat transfers occurring along the channel that must be taken into account in the thermal model [34,35]:

1. Longitudinal heat conduction inside the two active silicon layers, parallel to the microchannel.
2. Vertical heat conduction from the active silicon layers to surface of the top and bottom microchannel walls.
3. Vertical heat conduction between the active silicon layers through the microchannel silicon side walls.
4. Convective heat transfer from the surface of the microchannel walls into the bulk of the coolant.

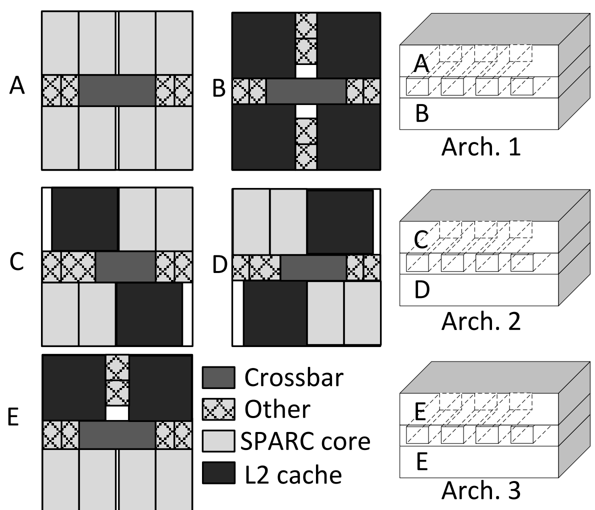


Fig. 2. Layout of the 3D-MPSoCs Used in our Experiments

5. Convective heat transport downstream along the channel due to the mass transfer (flow) of the coolant.

In our optimization, we define our cost function as the square of the Euclidean norm of the thermal gradient ($\mathbf{T}'$). Our optimal control design problem can be formulated as:

$$\min_{\mathbf{w_C}(z)} J = \int_0^d \|\mathbf{T}'\|^2 dz \tag{2}$$

Subject to : 1. System state-variable equations

2. Design constraints

3.2 Experimental Results

We apply the optimal channel modulation design to different liquid-cooled 3D-MPSoC architectures to demonstrate how the optimal channel modulation technique can be used with the conventional floorplan exploration to obtain the desired thermal behavior during the IC design. We use different configurations of the 90 nm UltraSPARC T1 (Niagara-1) processor [16] architecture. Fig. 2 shows the layout of the 3D-MPSoCs used in this experiment [30,16]. The dies are of size 1 cm $\times$ 1.1 cm and the heat flux densities range from 8 W/cm^2 to 64 W/cm^2.

In our optimization technique, we are using the worst-case (peak) power dissipation of the 3D-MPSoC functional elements [30,16]. Our proposed method achieves a thermal gradient reduction of 31% (23°C to 16°C). When the peak heat flux levels were replaced by average values, this same optimal channel modulation configuration manages to reduce the thermal gradient by 21 % compared

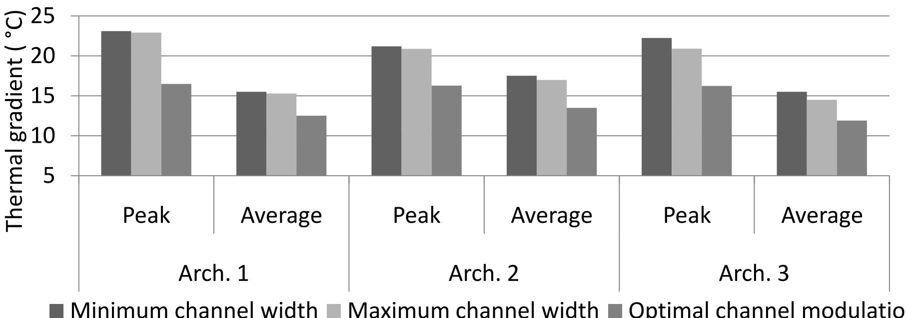


Fig. 3. Thermal Gradients Observed in the Different 3D-MPSoC Architectures Dissipating Peak and Average Level Heat Fluxes, Using Maximum, Minimum and Optimally Modulated Channel Widths

to the uniform channel width case. In addition, we observe that the peak temperature in the optimally modulated channel case equals to the peak temperature of the minimum channel width case. Thus, our proposal implicitly minimizes the peak temperature to the lowest value achievable within a given channel width range. The thermal gradients obtained for the different cases and for various channel types are plotted in Fig. 3. Sample thermal maps of the *Arch. 1* top-die, for the case of peak heat flux are also plotted in Fig. 4 to illustrate the ameliorating effect our proposed method has on the thermal gradients. The direction of coolant flow is from bottom to top of the figures.

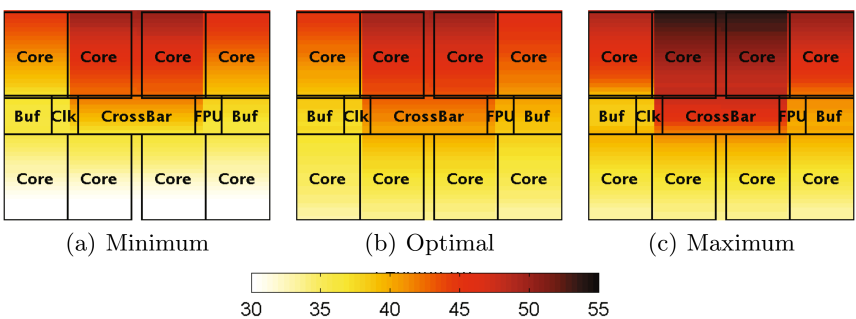


Fig. 4. Thermal Maps of Arch. 1 (Fig. 2) Top Die with Peak Heat Flux Levels, when Minimum, Maximum and Optimally Modulated Channel Widths are Applied. All the Thermal Maps are Drawn With Identical Temperature Scale ($[30 - 55]°C$)

4 Thermal-Aware Application Mapping on 3D Platforms

Distributing tasks optimally on a parallel platform is known to be NP-hard [9,40], but approximate methods exist. 3D platforms add new aspects to the problem and require rethinking the methods for system-level analysis, optimization, and

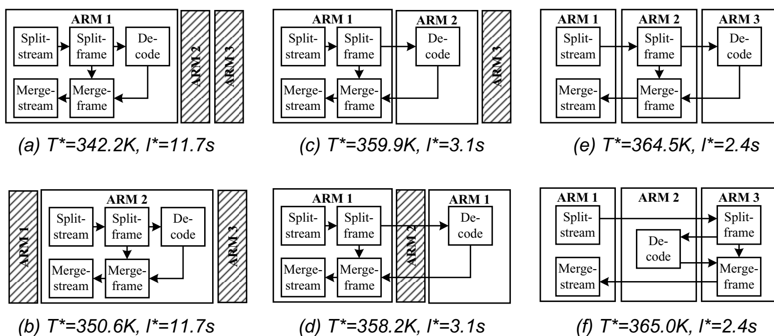


Fig. 5. Worst-case Latency Versus Worst-case Peak Temperature for Similar Bindings but Different Placements, of an MJPEG Decoder Evaluated on MPARM Platform [4]

exploration of the design space. Although considering thermodynamics of 3D stacks at system-level is crucial, none of the existing system-level mapping frameworks is thermally aware. Considering thermal management at system-level is important not only because of high cooling costs or the potential reliability problems if the circuit is not correctly designed, but also because latencies and other performance metrics might depend on temperature. In particular, if temperature variations are ignored, unpredictable runtime overheads or unexpected performance degradations might occur, e.g., due to reactive thermal mechanisms such as dynamic voltage and frequency scaling.

Let us consider the diagram in Figure 5 that has been first introduced in [19] and [33]. Solution pairs where only the placement of processing components is different are illustrated and indicate that physical placement cannot be ignored in temperature analysis, e.g., mappings (a) and (b) have the same latency, but their peak temperatures differ by more than 8 K. Therefore, even if the mapping is already predefined, the system designer might still reduce the temperature by selecting a different placement. The same is true for the opposite case, when designs might violate temperature thresholds if the physical placement has not been properly included in the system-level analysis. These experiments show that temperature distributions and temperature peaks are not easy to infer at system level, since they are governed by complex dependencies on the actual topology of the chip, its physical parameters, heat transfer rules, and accumulated bursts of jobs in applications' workloads that actually produce worst-case temperatures [27]. In fact, for any manycore design, without accurate worst-case chip temperature analysis tools included into system performance analysis, no guarantees can be given and mappings cannot be ruled out at system-level. To answer all these challenges, we have extended the distributed operation layer (DOL) [39] to consider system-level thermal-aware task to processors mapping.

4.1 Mapping Optimization Framework

The mapping optimization cycle implemented in the distributed operation layer [39] is illustrated in Fig. 6. In PRO3D, DOL considers parallel streaming

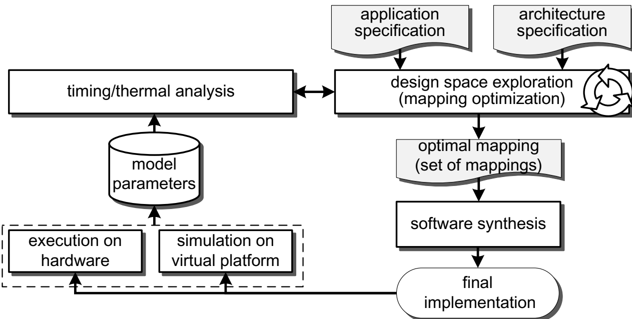


Fig. 6. Real-time and Thermal-aware Mapping Optimization Loop in Distributed Operation Layer for PRO3D (DOL3D)

applications represented as synchronous dataflow graphs (SDF) [15] and specified independently from the given PRO3D architecture. After the analysis of different design alternatives, a set of optimized mappings are provided. In PRO3D, each mapping is individually analyzed in terms of performance and (worst-case) thermal behavior. Finally, the chosen mapping specification will be further synthesized and implemented on the final system or can be simulated on the virtual platform. Typically, we use this low level simulation in a feedback loop for automatically calibrating the time and thermal analysis models [11,12,38].

4.2 Thermal Models and Analysis in DOL3D

Several system-level analysis models are included in DOL [39], ranging from very simple, static models to more complex, dynamic analytic models such as modular performance analysis (MPA). MPA [43] is an analytic approach targeting real-time systems and based on real-time calculus (RTC) [37]. From elementary knowledge about the best-case and worst-case behavior of system components in all operating conditions, MPA provides hard upper and lower bounds for various performance criteria of the system, such as end-to-end delays, buffer requirements, or resource utilizations. The system is abstractly modeled by bounded timing properties of event streams traversing the system, bounded capabilities of architectural units, and bounded execution requirements of event streams on individual components. Abstract components define the semantics of task executions and resource sharing mechanisms. Based on these abstractions, in classical timing analysis, the critical instant of task releases is used to guarantee the system worst-case execution time. Inspired from this time critical instant, we determine the temperature critical instant guaranteeing the worst-case peak temperature in the system in [27]. Similar to timing analysis, this critical temperature trace is identified among infinitely many traces that comply with the event stream specification in MPA and then the temperature of the system is simulated for the identified critical temperature trace. To apply the proposed method in [27] to a multi-core system such as PRO3D, in [33] we have extended the analysis to also consider the heat transfer among neighboring components.

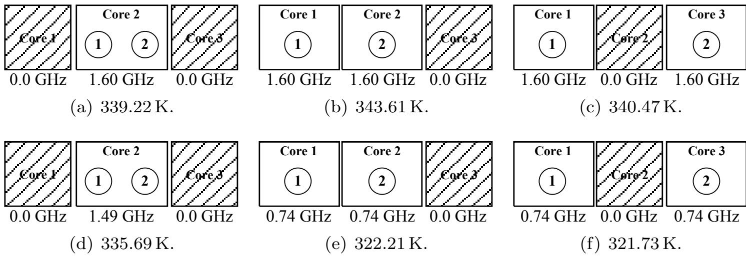


Fig. 7. Worst-Case Chip Temperature for Different Task Assignments and Clock Frequencies

Therefore, in [33] we provide a tight upper bound on the worst-case peak temperature of the entire multi-core system.

However, the method proposed in [33] uses linear search to calculate a tight bound on the worst-case peak temperature, and therefore exhibits a too long execution time for the design space exploration of a multi-core system with tens of processing components. An approximate method with a lower time complexity and that is three orders of magnitude faster has been determined in [32] to calculate an upper bound on the maximum temperature of a multi-core system. To extend the search options in the design space, in a thermal-aware task assignment is currently investigated such that the worst-case chip temperature is minimized and all real-time deadlines are met. This is possible due to individual static frequency selection for all cores in the system. An illustrative example is shown in Figure 7, where two identical tasks are mapped on three homogeneous processing components. When assigning the maximum operation frequency on all cores, the worst-case chip temperature is obtained when the tasks are assigned to different processing components. This is because both processing components process in parallel in the thermal critical scenario. When the operation frequency of every processing component is the minimum frequency such that all deadlines are just met, the lowest peak temperature is found when both tasks are mapped to different, non-adjoined processing components. This is because the individual operation frequencies can drastically be reduced when tasks are mapped onto different processing components.

The techniques described so far can be applied at design time, having the advantage of thermal-aware performance estimations and early thermal optimizations. However, in spite of thermal-aware design-time choices, there may be the need to respond to run-time thermal emergencies. In this case, specific thermal management actions might be applied as those described in section . However, to benefit of pre-calculated and still predictable performance, these dynamics have to be a-priori considered and included in the design strategy. One option is to select a set of optimized mappings after the design space exploration, instead of just one mapping. Each such mapping is having different guaranteed performance and temperature characteristics that can be exploited at run-time. The alternative is to apply control-theory to control the speed of processors in a loop receiving feedback from temperature sensors as described in [14]. The solution

in [14] is designed to meet thermal constraints and simultaneously provide safe bounds on worst-case delays suffered by all jobs in the system.

5 Generation and Simulation of the System-Model

The PRO3D system construction method [7] starts from a DOL [39,12] specification and is both rigorous and allows fine-grain analysis of system dynamics. It is rigorous because it is based on formal BIP models [3] with precise semantics that can be analyzed by using formal techniques. A system model in BIP is derived by progressively integrating constraints induced on an application software by the underlying hardware. It is obtained, in a compositional and incremental manner, from BIP models of the application software and respectively, the hardware platform, by application of source-to-source transformations that are proven correct-by-construction [7]. The system model describes the behavior of the mixed HW/SW system and can be simulated and formally verified using the BIP toolset.

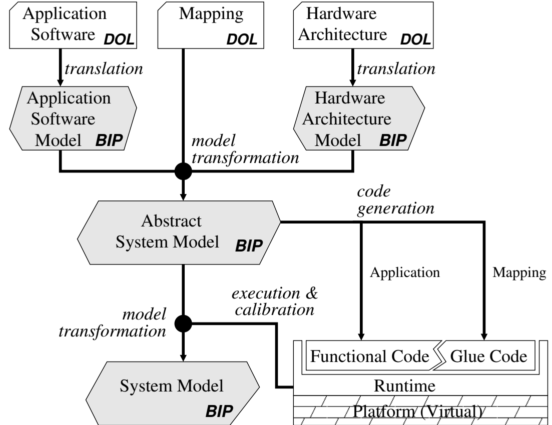


Fig. 8. System Model Construction & Code Generation

The method for the construction of mixed HW/SW system models is illustrated in Fig. 8. It takes as inputs: (i) the (untimed) application software, (ii) the (timed) hardware architecture and (iii) the mapping between them described in DOL. It proceeds in two main steps. The first step is the construction of the *abstract system model*. This model represents the behavior of the application software running on the hardware platform according to the mapping, but without taking into account execution times for the software actions. In the second step, the (bounds for) execution times are obtained by running every software process in isolation on the platform. These bounds are injected into the abstract system model and lead to the *system model*. This final model allows for the accurate estimation through simulation of real-time characteristics (response times,

delays, latencies, throughput, etc.) and indicators regarding resource usage (bus conflicts, memory conflicts, etc.).

System models are furthermore used for platform-dependent code generation. As illustrated in the Fig. 8, the generated code consists mainly of two parts: the *functional code*, which implements the different application tasks and their communication and the *glue code*, which implements the deployment of the application onto the platform according to the mapping and manages its execution lifecycle. This code is built on top of platform *runtimes*, that is, available APIs and libraries for thread management, memory allocation, communication and synchronization, etc. Once generated, the code is compiled by the native platform compiler and linked with the runtime libraries to produce the binary image for execution on the platform. This approach has been implemented and validated on *mpsim* (MPARM cycle-accurate simulator), *Gepop* (STHORM Posix simulator), STHORM TLM simulator and will be tested on the real STHORM silicon during Fall 2012. As for the target runtimes, we originally started using the Native Programming Layer (NPL), a common runtime implemented for both MPARM and STHORM; since mid-2012 we developed an implementation of the MCAPI standard for the STHORM platform [22].

6 STHORM, a Manycore Platform

Formerly known as P2012 [5,20,36] the STHORM modular architecture is shown Fig. 9. At the fabric level, an asynchronous NoC (Network-on-Chip) is organized in a 2D-mesh topology of clock-less routers. Each router has a NI (Network Interface) that connects to a cluster made of up to 16 cores in SMP and a number

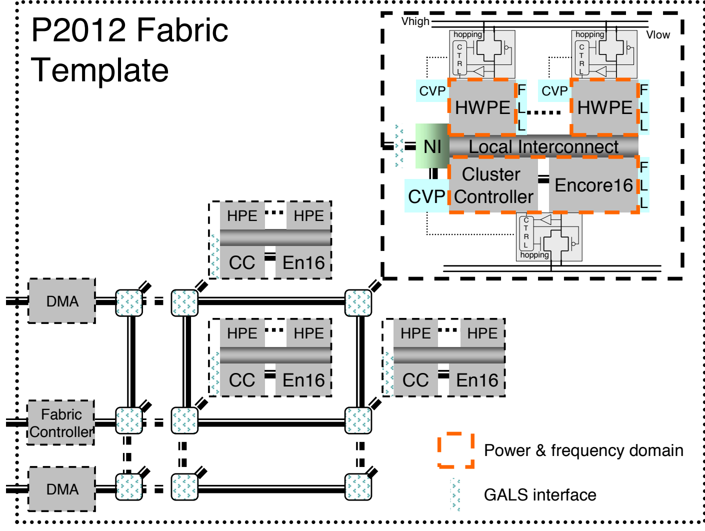


Fig. 9. The STHORM Computing Fabric Template

of communication engines to connect user defined HW IPs. This archtecture is a natural Globally Asynchronous Locally Synchronous (GALS) scheme and isolates logically the clusters. The NI gives access to the cluster main Clock, Variability and Power (CVP) controller, to control a power management harness. Within PRO3D we investigate 3-tier stacking for STHORM: a bottom SoC carrier for the general purpose host and IOs, a STHORM computing die, and a memory die. The experiments will include a number of VPI thermal modeling extensions to exercise the whole PRO3D SW development flow.

7 Conclusion after 30 Months into PRO3D

Two years and a half into its workplan, PRO3D has developed a number of tools and already assembled them into a consistent 3D exploration and programming workflow: A compact transient thermal model for simulation of 3D ICs with liquid cooling and its corresponding monitoring runtimes, a flexible virtual platform infrastructure for modelling and analysis of 3D architecture, a high level mapping optimisation tool focusing on peformance and preliminary support for temperature analyses, a rigorous transformation toolset for components that allow for the construction and assembly of system models and the generation of distributed intermediate format for deployment on the target platform. The last year of PRO3D will be focused on experiments with an actual industrial embedded manycore platform STHORM. Experiments have started on virtual platforms, and will move to real STHORM silicon during Fall of 2012.

Challenges for 3D and Programming

Besides these practical results, we think that the main challenges raised by 3D are a related to a retrofit of characteristics of the architecture into compilation flows and runtimes. Somehow, this is very similar to the issues encountered in HPC with distributed machines in the early '90. The problem is difficult, but a wide body of literature exists for purely topological issues. The new issues introduced by 3D stacking are mostly related to thermal aspects. Theses issues have two main origins:

1. *Thermal cross-coupling of execution units.* The relative position of processing units as a whole, or computing units from therein (operators, instructions decoders, register files, caches, etc.) and memory defines how heat from one element impacts another one. If two processors are too close to each other, we may have to offload both of them in situations where a single one could have run without harm. So not only the topoplogy of the manycore will have to be know from the compilation flow and the runtime, but also the geometry and thermal characteristics of the hardware [38];
2. *Different time scale for thermal propagation and computation forecast.* Manycore architecture are in the GHz range, while the evolution of the temperature is in the Hz range. This means several order of magnitude between the cause of heating –computations– and heating itself [34]. This gap in dynamic

magnitude is reinforced by the fact that even at constant frequency, energy consumption increases with temperature. All this makes it difficult to reverse temperature variations. Any decision related to thermal management will probably have to use predictive thermal models [1].

We thing that this will bring a number of consequence on programming models, compilation and runtimes:

- *The fading of pure static compilation.* Due to the huge gap of time scale between computation and thermal effects, it seems very difficult, if doable at all, to build full-static compilation schemes where the compiler will decide of the mapping off-line, before execution, once and for all. At least to ensure platform's thermal integrity, some level of responsibility w.r.t. mapping must be left to the runtime [44]. To ensure this integrity the runtime will have to deal with tasks scheduling and resource allocation while taking into account not only the architecture's topology and the computation load [18], but also the actual geometry and thermal characteristics of the material involved in the architecture [29]. This will require programming models that can provide enough flexibility at execution whereas essential properties can be guaranteed at compile-time [3].
- *The fading of von Neumann as a programming model.* As for programming models, we should move away from von Neumann –only as programming model, not as computing architecture– and consider other kinds of programming models naturally parallel, like process network and message passing already discussed [2,13,17]. Even these parallel programming models must be checked to be amendable to analyses that can predict the amount of computing load, if not to an absolute time reference, at least towards a moving horizon. This is necessary to provide computation forecasts to a runtime scheduler that can efficiently use the stacked architecture while preserving its thermal integrity.

Acknowledgments and Consortium. PRO3D is funded by the EU under FP7 GA n° 248776. It brings together **CEA**, Commissariat à l'énergie atomique et aux énergies alternatives (coord.), Fr.; **VERIMAG**, represented by Université Joseph Fourier Grenoble 1, Fr.; **ETHZ**, Eidgenössische Technische Hochschule Zürich, CH; **UNIBO**, Università di Bologna, It.; **STM**, STMicroelectronics, Fr.; **EPFL**, École polytechnique fédérale de Lausanne, CH. PRO3D Started in Jan. 2010 for an original duration of 30 months. It has been granted a six months extension to experiment with actual STHORM silicon, and will now end in Dec. 2012

References

1. Aly, S., Mostafa, M., Coskun, A.K., Atienza Alonso, D.: Fuzzy Control for Enforcing Energy Efficiency in High-Performance 3D Systems. In: Proceedings of the 2010 International Conference on Computer-Aided Design, ICCAD 2010, New York (2010)

2. Basu, A., Bozga, M., Sifakis, J.: Modeling Heterogeneous Real-time Systems in BIP. In: Software Engineering and Formal Methods SEFM 2006 Proceedings, pp. 3–12. IEEE Computer Society Press (2006)
3. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous component-based design using the BIP framework. IEEE Software, Special Edition – Software Components: Beyond Programming 28(3), 41–48 (2011)
4. Benini, L., Bertozzi, D., Bogliolo, A., Menichelli, F., Olivieri, M.: MPARM: Exploring the Multi-Processor SoC Design Space with SystemC. J. VLSI Signal Process. 41(2), 169–182 (2005)
5. Benini, L., Flamand, E., Fuin, D., Melpignano, D.: P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In: Rosenstiel, W., Thiele, L. (eds.) DATE 2012, pp. 983–987. IEEE (March 2012)
6. Bortolotti, D., Paterna, F., Pinto, C., Marongiu, A., Ruggiero, M., Benini, L.: Exploring instruction caching strategies for tightly-coupled shared-memory clusters. In: Int. Symp. on Systems-on-Chip (2011)
7. Bourgos, P., Basu, A., Bozga, M., Bensalem, S., Sifakis, J., Huang, K.: Rigorous system level modeling and analysis of mixed HW/SW systems. In: Proceedings of MEMOCODE, pp. 11–20. IEEE/ACM (2011)
8. Brunschwiler, T., et al.: Interlayer cooling potential in vertically integrated packages. Microsyst. Technol. 15(1), 57–74 (2009)
9. Burns, A.: Scheduling hard real-time systems: a review. Softw. Eng. J. 6, 116–128 (1991)
10. Coskun, A.K., et al.: Utilizing predictors for efficient thermal management in multiprocessor socs. IEEE Transactions on CAD 28(10), 1503–1516 (2009)
11. Haid, W., Keller, M., Huang, K., Bacivarov, I., Thiele, L.: Generation and calibration of compositional performance analysis models for multi-processor systems. In: Proc. Intl Conference on Systems, Architectures, Modeling and Simulation, SAMOS, pp. 92–99. IEEE, Samos (2009)
12. Huang, K., Haid, W., Bacivarov, I., Keller, M., Thiele, L.: Embedding formal performance analysis into the design cycle of MPSoCs for real-time streaming applications. ACM Trans. Embed. Comput. Syst. 11(1), 8:1–8:23 (2012), `http://doi.acm.org/10.1145/2146417.2146425`
13. Joven, J., Marongiu, A., Angiolini, F., Benini, L., De Micheli, G.: Exploring programming model-driven QoS support for noc-based platforms. In: 2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS, pp. 65–74 (October 2010)
14. Kumar, P., Thiele, L.: Timing analysis on a processor with temperature-controlled speed scaling. In: Proc. IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS. IEEE Computer, Beijing (2012)
15. Lee, E., Messerschmitt, D.: Synchronous data flow. Proceedings of the IEEE 75(9), 1235–1245 (1987)
16. Leon, A., et al.: A power-efficient high-throughput 32-thread SPARC processor. In: ISSCC, vol. 42(1), pp. 7–16 (2007)
17. Marongiu, A., Benini, L.: An OpenMP compiler for efficient use of distributed scratchpad memory in MPSoCs. IEEE Transactions on Computers PP(99), 1 (2010)
18. Marongiu, A., Burgio, P., Benini, L.: Vertical stealing: robust, locality-aware do-all workload distribution for 3D MPSoCs. In: Kathail, V., Tatge, R., Barua, R. (eds.) CASES, pp. 207–216. ACM (2010)

19. Marwedel, P., Teich, J., Kouveli, G., Bacivarov, J., Thiele, L., Ha, S., Lee, C., Xu, Q., Huang, L.: Mapping of applications to MPSoCs. In: 2011 Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS, pp. 109–118 (October 2011)
20. Melpignano, D., Benini, L., Flamand, E., Jego, B., Lepley, T., Haugou, G., Clermidy, F., Dutoit, D.: Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications. In: Groeneveld, P., Sciuto, D., Hassoun, S. (eds.) DAC, pp. 1137–1142. ACM (June 2012)
21. Micheli, G.D., Pavlidis, V., Alonso, D.A., Leblebici, Y.: Design methods and tools for 3D integration. In: Proceedings of the Symposium on VLSI Technology, Kyoto, Japan, pp. 182–183 (June 2011)
22. The Multicore Association: The Multicore Communications API (MCAPI™) v2.015 (2011), `http://www.multicore-association.org`
23. NVIDIA: Next Generation CUDA Compute Architecture: Fermi, whitepaper (2010), `http://www.nvidia.com`
24. Plurality: The HyperCore Processor. Plurality Ltd. (2010), `http://www.plurality.com`
25. PRO3D – Programming for Future 3D Multicore Architectures (2010), `http://pro3d.eu`
26. Qian, H., et al.: Cyber-physical thermal management of 3D multi-core cache-processor system with microfluidic cooling. ASP Journal of Low Power Electronics 7(1), 1–12 (2011)
27. Rai, D., Yang, H., Bacivarov, I., Chen, J.J., Thiele, L.: Worst-case temperature analysis for real-time systems. In: Design, Automation Test in Europe Conference Exhibition, DATE, pp. 1–6 (March 2011)
28. Ruggiero, M., Angiolini, F., Poletti, F., Bertozzi, D., Benini, L., Zafalon, R.: Scalability analysis of evolving SoC interconnect protocols. In: Int. Symp. on Systems-on-Chip, pp. 169–172 (2004)
29. Sabry, M.M., Atienza, D., Coskun, A.K.: Thermal Analysis and Active Cooling Management for 3D MPSoCs. In: Proceedings of IEEE International Symposium on Circuits and Systems, ISCAS 2011 (2011)
30. Sabry, M.M., et al.: Energy-Efficient Multi-Objective Thermal Control for Liquid-Cooled 3D Stacked Architectures. IEEE Transactions on CAD 30(12), 1883–1896 (2011)
31. Sabry, M.M., Ruggiero, M., Del Valle, P.G.: Performance and energy trade-offs analysis of L2 on-chip cache architectures for embedded MPSoCs. In: Proceedings of the 20th Symposium on Great Lakes Symposium on VLSI, GLSVLSI 2010, pp. 305–310. ACM, New York (2010)
32. Schor, L., Bacivarov, I., Yang, H., Thiele, L.: Fast worst-case peak temperature evaluation for real-time applications on multi-core systems. In: Proc. IEEE Latin American Test Workshop, LATW. IEEE, Quito (2012)
33. Schor, L., Bacivarov, I., Yang, H., Thiele, L.: Worst-case temperature guarantees for real-time applications on multi-core systems. In: Proc. IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS. IEEE Computer, Beijing (2012)
34. Sridhar, A., Vincenzi, A., Ruggiero, M., Brunschwiler, T., Atienza, D.: 3D-ICE: Fast compact transient thermal modeling for 3D ICs with inter-tier liquid cooling. In: 2010 IEEE/ACM International Conference on Computer-Aided Design, ICCAD, pp. 463–470 (2010)

35. Sridhar, A., Vincenzi, A., Ruggiero, M., Brunschwiler, T., Atienza, D.: Compact transient thermal model for 3D ICs with liquid cooling via enhanced heat transfer cavity geometries. In: 2010 16th International Workshop on Thermal Investigations of ICs and Systems, THERMINIC, pp. 1–6 (2010)
36. STMicroelectronics, CEA: Platform 2012 – A Manycore Programmable Accelerator for Ultra-Efficient Embedded Computing in Nanometer Technology (November 2010) (whitepaper)
37. Thiele, L., Chakraborty, S., Naedele, M.: Real-Time Calculus for Scheduling Hard Real-Time Systems. In: Proc. IEEE Int'l Symposium on Circuits and Systems, ISCAS, vol. 4, pp. 101–104 (2000)
38. Thiele, L., Schor, L., Yang, H., Bacivarov, I.: Thermal-aware system analysis and software synthesis for embedded multi-processors. In: Proc. Design Automation Conference, DAC, pp. 268–273. ACM, San Diego (2011)
39. Thiele, L., Bacivarov, I., Haid, W., Huang, K.: Mapping Applications to Tiled Multiprocessor Embedded Systems. In: Proc. Int'l Conf. on Application of Concurrency to System Design, ACSD, pp. 29–40 (2007)
40. Tindell, K.W., Burns, A., Wellings, A.J.: Allocating hard real-time tasks: an np-hard problem made easy. Real-Time Syst. 4, 145–165 (1992)
41. Tuckerman, D.B., Pease, R.F.W.: High-performance heat sinking for VLSI. IEEE Electron. Device Letters 5, 126–129 (1981)
42. Vincenzi, A., Sridhar, A., Ruggiero, M., Atienza, D.: Fast thermal simulation of 2D/3D integrated circuits exploiting neural networks and GPUs. In: Proceedings of the 17th IEEE/ACM International Symposium on Low-Power Electronics and Design, ISLPED 2011, pp. 151–156. IEEE Press, Piscataway (2011)
43. Wandeler, E., Thiele, L., Verhoef, M., Lieverse, P.: System architecture evaluation using modular performance analysis - a case study. Software Tools for Technology Transfer (STTT) 8(6), 649–667 (2006)
44. Zanini, F., Atienza, D., Benini, L., de Micheli, G.: Thermal-Aware System-Level Modeling and Management for Multi-Processor Systems-on-Chip. In: Proceedings of IEEE International Symposium on Circuits and Systems, ISCAS 2011 (2011)
45. Zervas, M., Temiz, Y., Leblebici, Y.: Fabrication and characterization of wafer-level deep tsv arrays. In: Proceedings of 2012 Electronic Components and Technology Conference, San Diego, CA (2012)

Thermal-Aware Task Assignment for Real-Time Applications on Multi-Core Systems

Lars Schor, Hoeseok Yang, Iuliana Bacivarov, and Lothar Thiele

Computer Engineering and Networks Laboratory,
ETH Zurich, 8092 Zurich, Switzerland
firstname.lastname@tik.ee.ethz.ch

Abstract. The reduced feature size of electronic systems and the demand for high performance lead to increased power densities and high chip temperatures, which in turn reduce the system reliability. Thermal-aware task allocation and scheduling algorithms are promising approaches to reduce the peak temperature of multi-core systems with real-time constraints. However, as long as the worst-case chip temperature is not incorporated into system analysis, no guarantees on the performance can be given. This paper explores thermal-aware task assignment strategies for real-time applications with non-deterministic workload that are running on a multi-core system. In particular, tasks are assigned to the multi-core system so that the worst-case chip temperature is minimized and all real-time deadlines are met. Each core has its own clock domain and the static assigned frequency corresponds to the minimum operation frequency such that no real-time deadline is missed. Finally, we show that the proposed temperature minimization problem can efficiently be solved by metaheuristics.

Keywords: Real-Time Systems, Worst-Case Chip Temperature, Task Assignment, Thermal Analysis, Multi-Core Systems.

1 Introduction

Multi-core systems outperform single-core platforms by offering higher performance and better power efficiency. However, the demand for increased performance and the reduced feature sizes lead to increasing power densities and high chip temperatures, which in turn reduce the system reliability. For example, exceeding the chip's peak temperature could lead to a reduction of performance or even damage the physical system. Reactive thermal management mechanisms, cooling systems, and thermal-aware task allocation and scheduling algorithms are potential techniques to tackle thermal and reliability issues.

Cooling systems for embedded real-time systems have to be designed for the worst-case chip temperature, i.e., the maximum chip temperature under all feasible scenarios of task arrivals. As the packaging costs of cooling systems increase super-linearly in power consumption [1], its design might be very expensive without the use of other thermal management mechanisms. Reactive thermal management mechanisms such as DVFS [2, 3] are widely used to address thermal

B. Beckert et al. (Eds.): FMCO 2011, LNCS 7542, pp. 294–313, 2013.
© Springer-Verlag Berlin Heidelberg 2013

issues. Despite their thermal effectiveness, these techniques cause a significant degradation of performance or lead to an expensive run-time overhead, both unacceptable in today's embedded real-time systems.

Various thermal-aware task allocation and scheduling algorithms have recently been studied [4–7]. However, as long as the worst-case chip temperature is not incorporated into system analysis, no guarantees on performance can be given and violations of real-time deadlines cannot be ruled out. Consequently, this paper explores thermal-aware task assignment and frequency selection strategies to reduce the worst-case chip temperature under real-time constraints. In particular, we consider the following problem:

Given are a set of tasks that are mapped onto a multi-core chip. Then, the goal is to assign each processing component its optimal frequency and to select a static assignment of tasks to processing components such that all real-time deadlines are met and the worst-case chip temperature is minimized.

To this end, we propose a thermal analysis method to calculate a non-trivial upper bound on the maximum temperature of an embedded real-time system with multiple cores and non-deterministic workload that is later incorporated into the task assignment problem. Arrival curves from real-time calculus [8] are used to upper bound the task's workload in any time interval. Each processing component executes at a static frequency assigned at compile-time. This frequency is selected such that the real-time deadlines of all tasks are met and the worst-case chip temperature is minimized. The considered thermal model is able to address various thermal effects like the heat exchange between neighboring cores and temperature-dependent leakage power. The contributions of this paper can be summarized as follows:

- A novel method to calculate the worst-case chip temperature of an embedded real-time system with multiple cores and non-deterministic workload is formally derived.
- The minimization of the worst-case chip temperature with respect to real-time constraints is formulated as a nonlinear binary integer problem.
- We show the viability of the proposed methods in various case studies on hardware platforms with up to 16 cores.

The remainder of the paper is organized as follows: First, the considered problem is motivated by an introductory example in Section 2. Afterwards, in Section 3, the thermal and computational models considered in this paper are introduced. In Section 4, we discuss a method to calculate the worst-case chip temperature and show how to select the optimal operation frequency. The optimal task assignment problem is formulated as a nonlinear binary integer problem in Section 5. Finally, Section 6 presents case studies to highlight the viability of our methods and related work is discussed in Section 7.

2 Motivational Example

System Description. In order to motivate the considered problem, we examine various task to processing component assignments of a simple system with two identical tasks ν_1 and ν_2, and three homogeneous processing components with a maximum operation frequency of 1.6 GHz. The chip floorplan corresponds to the one outlined in Fig. 2. The parameters of the thermal model and the power dissipation parameters are summarized in Table 1(a) and Table 1(b). Both tasks have an invocation interval of 200 ms, a jitter of 400 ms, and a computational demand of $5 \cdot 10^7$ cycles, i.e., 31.25 ms when the processing component is running at its maximum operation frequency. Furthermore, the real-time deadline of a task is equal to its period.

Maximum Operation Frequency. First, we suppose that each processing component can only process at its maximum operation frequency, i.e., 1.6 GHz. Then, we calculate the worst-case chip temperature, i.e., the maximum chip temperature under all feasible scenarios of task arrivals for different mappings by the method proposed in Section 4.1. Mapping both tasks ν_1 and ν_2 to the same processing component results in a worst-case chip temperature of 339.22 K while mapping both tasks to different adjoined and non-adjoined processing components leads to a worst-case chip temperature of 343.61 K and 340.47 K, respectively. Note that the worst-case chip temperature is higher when both tasks are assigned to different processing components as both processing components are concurrently processing in the thermal critical scenario.

Optimal Operation Frequency. Next, the operation frequency of every processing component is the minimum frequency such that all deadlines are just met, calculated by (21). When both tasks are mapped onto the same processing component, the frequency can be reduced to 1.49 GHz and the maximum temperature of the system is 335.69 K. Assigning both tasks to different adjoined and non-adjoined processing components results in a worst-case chip temperature of 322.21 K and 321.73 K, respectively. As the operation frequency can be reduced to 0.74 GHz when both tasks are mapped onto different processing components, we observe the lowest worst-case chip temperature when both tasks are mapped onto non-adjoined processing components. In particular, the worst-case chip temperature was reduced by almost 22 K by selecting an adequate task to processing component assignment and optimal operation frequencies.

3 System Model and Problem Definition

In this section, the task, power, and temperature models are described.

Notation: Bold characters will be used for vectors and matrices, and non-bold characters will be used for scalars. For example, $\mathbf{H}$ denotes a matrix whose (k, ℓ)-th element is denoted as $H_{k\ell}$ and $\mathbf{T}$ denotes a vector whose k-th element is denoted as T_k.

3.1 Task Model

The task model considered in this paper is based on real-time calculus [8]. Let $\boldsymbol{\nu}$ be the set of tasks that are executed. We suppose that task ν_j is a stream of events and has a total workload of $R_{\nu_j}(s,t)$ cycles in time interval $[s,t)$. Each event has to complete its execution within D_{ν_j} time units after its arrival. The arrival curve α_{ν_j} upper bounds all possible cumulative workloads:

$$R_{\nu_j}(s,t) \leq \alpha_{\nu_j}(t-s) \quad \forall 0 \leq s < t \tag{1}$$

with $\alpha_{\nu_j}(\Delta) = 0$ for all $\Delta \leq 0$. In other words, $R_{\nu_j}(0,t)$ is the cumulative number of computing cycles of all events arrived in $[0,t)$. Arrival curves are a generalization of various well-know event arrival models as, for example, periodic event arrivals with jitter [9].

We use the concept of a demand bound function [10] to model the maximum resource demand of a task, and later to check the schedulability. In particular, the demand bound function $\mathrm{dbf}_{\nu_j}(\Delta)$ of task ν_j is:

$$\mathrm{dbf}_{\nu_j}(\Delta) = \alpha_{\nu_j}(\Delta - D_{\nu_j}) \quad \forall \Delta \geq 0\ . \tag{2}$$

In other words, the maximum accumulated computational demand of all events that arrive and have deadline in any interval of length Δ does not exceed $\mathrm{dbf}_{\nu_j}(\Delta)$.

3.2 Processor Model

We consider a homogeneous multi-core system with a set of processing components $\boldsymbol{\Theta}$. The total accumulated workload of Θ_ℓ at time t is denoted as $R_\ell(0,t)$ and is, in any time interval of length $\Delta \leq t$, upper bounded by the arrival curve α_ℓ [8]:

$$R_\ell(t-\Delta,t) \leq \alpha_\ell(\Delta) = \sum_{j=1}^{|\boldsymbol{\nu}|} \Gamma(\nu_j,\Theta_\ell) \cdot \alpha_{\nu_j}(\Delta) \tag{3}$$

with the assignment function:

$$\Gamma(\nu_j,\Theta_\ell) = \begin{cases} 1 & \text{if } \nu_j \text{ executes on processing component } \Theta_\ell \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

Likewise, the demand bound function $\mathrm{dbf}_\ell(\Delta)$ of a processing component Θ_ℓ can be calculated. For example, suppose that an earliest-deadline-first (EDF) scheduler runs on each processing component to arbitrate between events of different tasks assigned to the same processing component. Then, the demand bound function $\mathrm{dbf}_\ell(\Delta)$ is [10]:

$$\mathrm{dbf}_\ell(\Delta) = \sum_{j=1}^{|\boldsymbol{\nu}|} \Gamma(\nu_j,\Theta_\ell) \cdot \mathrm{dbf}_{\nu_j}(\Delta)\ . \tag{5}$$

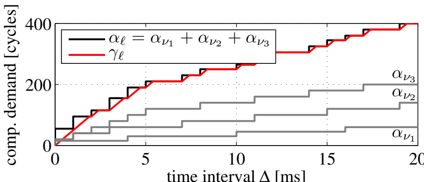


Fig. 1. Typical arrival curve $\alpha_\ell(\Delta)$ with its corresponding upper bound on the accumulated computing time $\gamma_\ell(\Delta)$

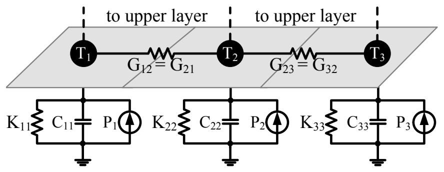


Fig. 2. *RC* circuit of the silicon layer for a chip with three processing components

Each processing component executes at a static frequency f_ℓ with $0 < f_\ell \leq f_\ell^{\max}$. Therefore, the cumulated number of available computing cycles in time interval $[s, t)$ is $W_\ell(s, t) = f_\ell \cdot (t - s)$. If there are no waiting or arriving tasks in $[s, t)$, the available resources $W_\ell(s, t)$ are wasted. Otherwise, they are used to process incoming and waiting events.

The accumulated computing time $Q_\ell(0, t)$ describes the amount of cycles that processing component Θ_ℓ is spending to process an incoming workload of $R_\ell(0, t)$ time units. Using arrival curve $\alpha_\ell(\Delta)$, the accumulated computing time $Q_\ell(t - \Delta, t)$ can be upper bounded by $\gamma_\ell(\Delta)$ for all intervals of length $\Delta \leq t$ [11]:

$$Q_\ell(t - \Delta, t) \leq \gamma_\ell(\Delta) = \inf_{0 \leq \lambda \leq \Delta} \{\alpha_\ell(\lambda) + W_\ell(0, \Delta - \lambda)\} \quad . \tag{6}$$

From the properties of the arrival curve, it follows that $\gamma_\ell(\Delta)$ is monotonically increasing. The operation mode of a component can be expressed by the mode function $S_\ell(t)$, which is $S_\ell(t) = 1$ if the component is in 'active' processing mode at time t and $S_\ell(t) = 0$ if the component is in 'idle' processing mode at time t:

$$S_\ell(t) = \frac{\mathrm{d}Q_\ell(0, t)}{\mathrm{d}t} \cdot \frac{1}{f_\ell} = \begin{cases} 1 & \Theta_\ell \text{ is processing some events at time } t \\ 0 & \Theta_\ell \text{ is 'idle' at time } t. \end{cases} \tag{7}$$

A typical arrival curve and its corresponding upper bound on the accumulated computing time are outlined in Fig. 1.

The results of the paper also hold for heterogeneous platforms, but the task model becomes more complex. The workload $R_{\nu_j}(s, t)$ would just specify the number of events in time interval $[s, t)$. To calculate the accumulated computing time $Q_\ell(s, t)$, the workload of task ν_j is multiplied by the computation-time in cycles of an event of task ν_j when ν_j is assigned to processing component Θ_ℓ.

3.3 Power Model

The power consumption of a processing component is the sum of the dynamic and leakage power consumption [4, 12]. Whenever a component is processing some events, the component is in 'active' mode, and consumes both dynamic and leakage power. Otherwise, it is in 'idle' mode, and consumes only leakage power.

Each processing component Θ_ℓ has its own clock domain and we suppose that the dynamic power consumption $P_{\ell,\mathrm{dyn}}$ of component Θ_ℓ growths quadratically with its supply voltage v_ℓ and linearly with its operation frequency f_ℓ [13]:

$$P_{\ell,\mathrm{dyn}}(t) \propto v_\ell^2 \cdot f_\ell \cdot S_\ell(t) \tag{8}$$

where the mode function $S_\ell(t)$ implies that $P_{\ell,\mathrm{dyn}}(t) = 0$ if the component is in 'idle' processing mode. Note that the results of the paper also hold for other relations between supply voltage and frequency as long as they are monotone.

The leakage power consumption $P_{\ell,\mathrm{leak}}$ of component Θ_ℓ is super linearly dependent on the temperature that can approximately be modeled by a linear function of the temperature [6, 14]:

$$P_{\ell,\mathrm{leak}}(t) = \phi_{\ell\ell} \cdot T_\ell(t) + \psi_\ell \tag{9}$$

with T_ℓ the temperature of processing component ℓ, and the constants $\phi_{\ell\ell}$ and ψ_ℓ. We assume that the square of the supply voltage scales linearly with the operation frequency [5] and therefore, the total power consumption is:

$$\mathbf{P}(t) = \mathbf{P}_{\mathrm{dyn}}(t) + \mathbf{P}_{\mathrm{leak}}(t) = \boldsymbol{\phi} \cdot \mathbf{T}(t) + \boldsymbol{\rho} \cdot \mathrm{diag}(\mathbf{f})^3 \cdot \mathbf{S}(t) + \boldsymbol{\psi} \tag{10}$$

with the diagonal matrix $\mathrm{diag}(\mathbf{f})$ of vector $\mathbf{f}$ and a constant diagonal matrix $\boldsymbol{\rho}$.

3.4 Temperature Model

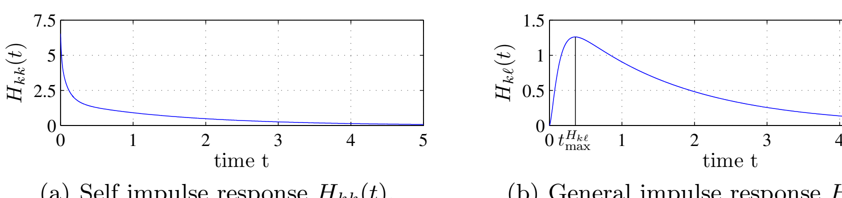
We model the temperature evolution of a multi-core system by an equivalent RC circuit [4, 15–17]. The vertical layout of the chip is modeled by four layers, namely the heat sink, heat spreader, thermal interface, and silicon die. Each layer is divided into a set of blocks according to architecture-level units, i.e., processing components, see Fig. 2 for the RC circuit of the silicon layer for a chip with three processing components. Every block is then mapped onto a node of the thermal circuit. The number of nodes and therefore, the order of the thermal model is $n = 4 \cdot |\Theta|$. In particular, the n-dimensional temperature vector $\mathbf{T}(t)$ at time t is described by a set of first-order differential equations:

$$\mathbf{C} \cdot \frac{\mathrm{d}\mathbf{T}(t)}{\mathrm{d}t} = \left(\mathbf{P}(t) + \mathbf{K} \cdot \mathbf{T}^{\mathrm{amb}}\right) - (\mathbf{G} + \mathbf{K}) \cdot \mathbf{T}(t) \tag{11}$$

with the $n \times n$ thermal capacitance matrix $\mathbf{C}$, the $n \times n$ thermal conductance matrix $\mathbf{G}$, the $n \times n$ thermal ground conductance matrix $\mathbf{K}$, the n-dimensional power dissipation vector $\mathbf{P}$, and the ambient temperature vector $\mathbf{T}^{\mathrm{amb}} = T^{\mathrm{amb}} \cdot [1, \dots, 1]'$. The initial temperature vector is denoted as $\mathbf{T}^0$ and the system is assumed to start at time $t^0 = 0$.

Rewriting (11) with (10) leads to the state-space representation of the thermal model:

$$\frac{\mathrm{d}\mathbf{T}(t)}{\mathrm{d}t} = \mathbf{A} \cdot \mathbf{T}(t) + \mathbf{B} \cdot \mathbf{u}(t) \tag{12}$$



(a) Self impulse response $H_{kk}(t)$.

(b) General impulse response $H_{k\ell}(t)$.

Fig. 3. Impulse responses

with input vector $\mathbf{u}(t) = \boldsymbol{\rho} \cdot \text{diag}(\mathbf{f})^3 \cdot \mathbf{S}(t) + \boldsymbol{\psi} + \mathbf{K} \cdot \mathbf{T}^{\text{amb}}$, $\mathbf{A} = -\mathbf{C}^{-1} \cdot (\mathbf{G} + \mathbf{K} - \boldsymbol{\phi})$, and $\mathbf{B} = \mathbf{C}^{-1}$. As the thermal system is linear and time-invariant, the temperature of node k is:

$$T_k(t) = T_k^{\text{init}}(t) + \sum_{\ell=1}^{n} T_{k,\ell}(t) \tag{13}$$

with $\mathbf{T}^{\text{init}}(t) = e^{\mathbf{A}\cdot t} \cdot \mathbf{T}^0$. $T_{k,\ell}(t)$ is the convolution of input u_ℓ and $H_{k\ell}$, i.e., the impulse response between nodes ℓ and k:

$$T_{k,\ell}(t) = \int_0^t H_{k\ell}(\xi) \cdot u_\ell(t - \xi)\, \mathrm{d}\xi \tag{14}$$

with

$$u_\ell(t) = \rho_{\ell\ell} \cdot f_\ell^3 \cdot S_\ell(t) + \psi_\ell + K_{\ell\ell} \cdot T^{\text{amb}} = \rho_{\ell\ell} \cdot f_\ell^3 \cdot S_\ell(t) + u_\ell^{\text{idle}} \ . \tag{15}$$

Nodes that do not correspond to a processing component have input $u_\ell = u_\ell^{\text{idle}} = \psi_\ell + K_{\ell\ell} \cdot T^{\text{amb}}$. Similar to [17], we assume that $H_{k\ell}(t)$ is a non-negative unimodal function that has its maximum at time $t_{\max}^{H_{k\ell}}$, see Fig. 3 for an illustration.

4 System Analysis

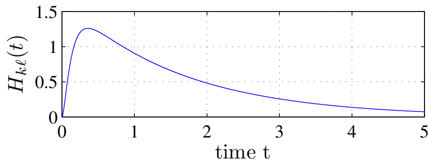
In this section, we propose a novel method to calculate the worst-case chip temperature of a multi-core system with non-deterministic workload and show how to select the operation frequencies in an optimal manner.

4.1 Peak Temperature Analysis

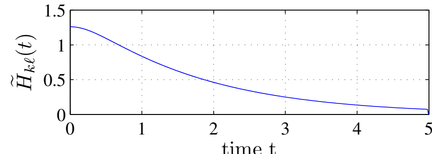
Suppose that the thermal RC network of a multi-core system is composed of n nodes. Then, the worst-case chip temperature T_S^* of a multi-core system is the maximum temperature of all individual nodes:

$$T_S^* = \max\left(T_1^*, \ldots, T_n^*\right) \tag{16}$$

where T_k^* is the worst-case peak temperature of node k. The following theorem follows from the results of [17] and states that the worst-case peak temperature is composed of $n+1$ summands, that can be calculated individually.



(a) Original impulse response $H_{k\ell}(t)$.



(b) Non-increasing left-continuous sorting $\widetilde{H}_{k\ell}(t)$.

Fig. 4. Example of a typical impulse response $H_{k\ell}(t)$ and its non-increasing left-continuous sorting equivalent $\widetilde{H}_{k\ell}(t)$

Theorem 1. *Suppose that* $T^*_{k,\ell}(\tau) = \max_{u_\ell \in U_\ell}(T_{k,\ell}(\tau))$ *with* U_ℓ *the set of all possible inputs* u_ℓ*,* $T_{k,\ell}(t)$ *defined as in (14), and a certain time instance* τ*. Then, an upper bound on the maximum temperature of node* k *at time* τ *is:*

$$T^*_k(\tau) \leq T^{\text{init}}_k + \sum_{\ell=1}^{n} T^*_{k,\ell}(\tau) \tag{17}$$

where n *is equal the number of nodes of the thermal RC circuit.*

Proof. Rewriting (13) with $T^*_{k,\ell}(\tau) = \max_{u_\ell \in U_\ell}(T_{k,\ell}(\tau))$ leads to:

$$\begin{aligned} T^*_k(\tau) &= \max_{\mathbf{u}\in\mathbf{U}}(T_k(\tau)) = \max_{\mathbf{u}\in\mathbf{U}}\left(T^{\text{init}}_k + \sum_{\ell=1}^{n} T_{k,\ell}(\tau)\right) \\ &\leq T^{\text{init}}_k + \sum_{\ell=1}^{n} \max_{u_\ell\in U_\ell}(T_{k,\ell}(\tau)) = T^{\text{init}}_k + \sum_{\ell=1}^{n} T^*_{k,\ell}(\tau). \quad \square \end{aligned} \tag{18}$$

As $T^*_{k,\ell}$ only depends on the workload of Θ_ℓ, we can individually maximize $T_{k,\ell}(\tau)$ at time instance τ for each processing component Θ_ℓ. The remaining question is how to calculate an upper bound $T^*_{k,\ell}(\tau)$ on $T_{k,\ell}(\tau)$ for a given time instance τ and all possible input sequences u_ℓ. To this end, we first introduce the non-increasing left-continuous sorting $\widetilde{H}_{k\ell}(t)$ of $H_{k\ell}(t)$ [18]. Roughly speaking, $\widetilde{H}_{k\ell}(t)$ is $H_{k\ell}(t)$ sorted in non-increasing order, see Fig. 4 for an example of a typical impulse response $H_{k\ell}(t)$ and its sorted equivalent $\widetilde{H}_{k\ell}(t)$. For illustration, we suppose discrete time, i.e., $H_{k\ell}(t)$ may change values only at multiples of δ and is constant for $t \in [r\cdot\delta, (r+1)\cdot\delta)$ for all $r \geq 0$. Then, $\widetilde{H}_{k\ell}[r]$ has the same elements as $H_{k\ell}[r]$, however, they are ordered non-increasingly, i.e., $\widetilde{H}_{k\ell}[r] \geq \widetilde{H}_{k\ell}[r+1]$ for all $r \geq 0$.

The next theorem shows that $T^*_{k,\ell}(\tau)$ is obtained by first calculating the sorted equivalent $\widetilde{H}_{k\ell}(t)$ of $H_{k\ell}(t)$, and then by convoluting $\widetilde{H}_{k\ell}(t)$ with $S^*_\ell(t)$. $S^*_\ell(t)$ is the mode function defined as in (7) resulting from the accumulated computing time $Q^*_\ell(0,t) = \gamma_\ell(\tau) - \gamma_\ell(\tau - t)$ for all $0 \leq t \leq \tau$. $Q^*_\ell(0,t)$ is called the thermal critical accumulated computing time. In particular, $Q^*_\ell(0,t)$ shifts the computing

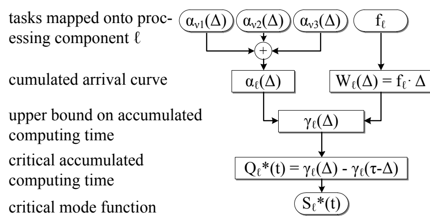


Fig. 5. Steps to calculate the critical accumulated computing time $Q_\ell^*(0,t)$ and the critical mode function $S_\ell^*(t)$.

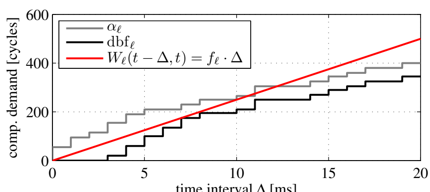


Fig. 6. Example of calculating the minimum operation frequency of component ℓ. $\alpha_\ell(\Delta)$ is the arrival curve defined as in Fig. 1, $\mathrm{dbf}_\ell(\Delta)$ the demand bound function, and $W_\ell(t-\Delta,t) = f_\ell \cdot \Delta$ the resulting available computing resources.

time as late as possible to observation time τ. The steps required to calculate $Q_\ell^*(0,t)$ are detailed in Section 3 and summarized in Fig. 5.

Theorem 2. *Suppose that $\widetilde{H}_{k\ell}(t)$ is the non-increasing left-continuous sorting of $H_{k\ell}(t)$ [18], $Q_\ell^*(0,t) = \gamma_\ell(\tau) - \gamma_\ell(\tau - t)$ for all $0 \leq t \leq \tau$, and $T_{k,\ell}(t)$ is defined as in (14). Then, for any given time instance τ, $T_{k,\ell}^*(\tau)$ defined as:*

$$T_{k,\ell}^*(\tau) = u_\ell^{\mathrm{idle}} \cdot \int_0^\tau H_{k\ell}(t-\xi)\,\mathrm{d}\xi + \rho_{\ell\ell} \cdot f_\ell^3 \cdot \int_0^\tau S_\ell^*(\xi) \cdot \widetilde{H}_{k\ell}(\tau - \xi)\,\mathrm{d}\xi \tag{19}$$

with $S_\ell^*(t) = \frac{\mathrm{d}Q_\ell^*(0,t)}{\mathrm{d}t}$ is an upper bound on $T_{k,\ell}(\tau)$, i.e., $T_{k,\ell}^*(\tau) \geq T_{k,\ell}(\tau)$.

Proof. The proof of this theorem is in the Appendix. □

So far, we have shown how to calculate an upper bound on the maximum temperature $T_k^*(\tau)$ of processing component k at time τ. However, we did not dwell on the amount of the observation time τ. The following theorem states that increasing the observation time τ will not decrease the worst-case peak temperature if $\mathbf{T}^0 \leq (\mathbf{T}^\infty)^i$, where $(\mathbf{T}^\infty)^i$ is the steady-state temperature vector if all components are in 'idle' processing mode.

Theorem 3. *Suppose that $T_k^*(\tau)$ defined as in (17) is an upper bound on the maximum temperature of processing component k at time τ. Then, $T_k^*(\tau) \geq T_k(t)$ for all $0 \leq t \leq \tau$ and for any set of feasible workload traces with the same initial temperature vector $\mathbf{T}^0 \leq (\mathbf{T}^\infty)^i$.*

Proof. With $T_{k,\ell}^*(\tau)$ defined as in (19), the proof is equivalent to the proof of Lemma 6 in [17]. □

In summary, Theorems 1 to 3 form together a method to calculate a non-trivial upper bound on the maximum temperature of a multi-core chip. First, we individually calculate $T_{k,\ell}^*(\tau)$ for all k, ℓ by (19). Then, in a second step, the maximum temperature $T_k^*(\tau)$ of each node k is calculated by (17). Finally, the

worst-case chip temperature T_S^* follows from (16). The proposed method provides a safe bound on the maximum temperature, i.e., the method guarantees that the actual chip temperature will never exceed the temperature T_S^*.

4.2 Optimal Frequency Assignment

As a frequency reduction always leads to an accumulated computing time that is in all time intervals $\Delta \geq 0$ smaller or equal the original accumulated computing time, a frequency reduction results in a lower worst-case chip temperature. Therefore, the optimal frequency of every processing component Θ_ℓ is the minimum operation frequency such that no real-time deadline is missed. In particular, Θ_ℓ is schedulable, i.e., the real-time deadlines of all events are met, if the cumulated number of available computing resources W_ℓ is in no time interval Δ smaller than the maximum resources demand dbf_ℓ defined as in (5) [10]:

$$\text{dbf}_\ell(\Delta) \leq W_\ell(t - \Delta, t) = f_\ell \cdot \Delta \quad \forall \Delta \geq 0 \ . \tag{20}$$

Therefore, the minimum operation frequency f_ℓ of processing component Θ_ℓ, such that all real-time deadlines are met, is:

$$f_\ell = \sup_{\Delta \geq 0} \left\{ \frac{\text{dbf}_\ell(\Delta)}{\Delta} \right\} \ . \tag{21}$$

In other words, the frequency is selected such that the computing resource curve $W_\ell(t - \Delta, t) = f_\ell \cdot \Delta$ upper bounds the maximum resources demand $\text{dbf}_\ell(\Delta)$ in every time interval $\Delta \geq 0$. From a geometric point of view, the problem is equivalent to determine a tangent to the curve $\text{dbf}_\ell(\Delta)$ that crosses the origin. Practically, the optimal frequency can be calculated by the RTC toolbox [19]. Figure 6 illustrates this calculation with the help of an example.

5 Optimal Task Assignment

In this section, the optimal task assignment formulation is stated that solves the problem defined in Section 1.

5.1 Temperature Minimization Problem

In the last section, we proposed a method to calculate the optimal operation frequency of each processing component. Now, we apply these results to calculate an optimal task assignment that minimizes the worst-case chip temperature and guarantees that all real-time deadlines are met. If the worst-case chip temperature is smaller than the critical chip temperature, the system can safely execute the assignment without involving other (dynamic) thermal management strategies, that may lead to unpredictable behavior.

The objective of the temperature minimization problem (TMP) is to reduce the worst-case chip temperature:

$$\text{minimize } T_S^* = \max\left(T_1^*, \ldots, T_n^*\right) \tag{22}$$

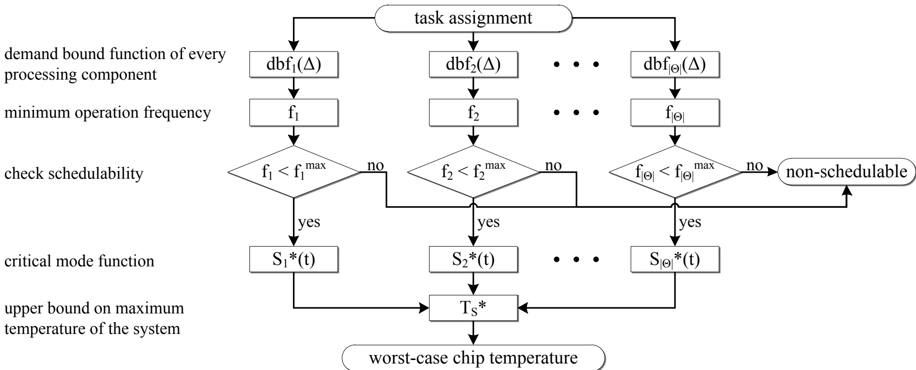


Fig. 7. Overview of the flow to analyze a single task assignment for schedulability and worst-case chip temperature

where $T_k^* \geq T_k(t)$ for all $t \geq 0$ is the worst-case peak temperature of node k and n the number of nodes of the thermal RC circuit of the chip.

Furthermore, the operation frequency f_ℓ of processing component Θ_ℓ has to fulfill the following constraints following from (2) , (5) and (21):

$$f_\ell = \sup_{\Delta \geq 0} \left\{ \frac{\sum_{j=1}^{|\boldsymbol{\nu}|} \Gamma(\nu_j, \Theta_\ell) \cdot \alpha_{\nu_j}(\Delta - D_{\nu_j})}{\Delta} \right\} \leq f_\ell^{\max} \quad . \tag{23}$$

This constraint is also used to guarantee the schedulability. Whenever the operation frequency f_ℓ is smaller or equal to the maximum frequency $f_\ell^{\max}$ of processing component ℓ, the considered task assignment is schedulable, and otherwise, the task assignment is infeasible.

Finally, we have to guarantee that all tasks are assigned to exactly one processing component:

$$\sum_{\ell=1}^{|\boldsymbol{\Theta}|} \Gamma(\nu_j, \Theta_\ell) = 1 \quad \forall \nu_j \in \boldsymbol{\nu} \quad . \tag{24}$$

5.2 Evaluating a Task Assignment

So far, we formulated the TMP as a nonlinear binary optimization problem. Next, we will describe how to apply the methods presented in Section 4 to verify the schedulability and, if the task assignment is feasible, to calculate an upper bound on the maximum chip temperature of a task assignment.

The proposed method is summarized in Fig. 7. First, the demand bound function related to every processing component is individually computed by (5). Then, the minimum operation frequency f_ℓ is calculated for all processing components Θ_ℓ by (21) and the schedulability is tested. In particular, the system is only schedulable if all frequencies f_ℓ are smaller or equal their maximum frequency $f_\ell^{\max}$. Once we know that the system is schedulable with a particular set

of operation frequencies, we calculate the worst-case chip temperature. To this end, the critical mode function S_ℓ^* is calculated for all processing components Θ_ℓ by the approach shown in Fig. 5. Finally, the worst-case chip temperature T_S^* follows by (16).

5.3 Efficient Temperature Reevaluation

Calculating the optimal task assignment might be expensive when all steps described in Section 5.2 are repeated for every possible assignment. Next, we show how to efficiently calculate the worst-case chip temperature of multiple related task assignments as it is the case in many metaheuristics. In a first step, we rewrite the formula to calculate an upper bound on the maximum temperature of a node k as two terms, thereof one is assignment-independent, i.e., reusable, and one is assignment-dependent. Rewriting (17) with (19) leads to:

$$T_k^*(\tau) \leq T_k^{\text{init}} + \sum_{\ell=1}^{n} \left(u_\ell^{\text{idle}} \cdot \int_0^\tau H_{k\ell}(t-\xi)\,\mathrm{d}\xi \right) + \sum_{\ell=1}^{n} \left(\rho_{\ell\ell} \cdot f_\ell^3 \cdot \int_0^\tau S_\ell^*(\xi) \cdot \widetilde{H}_{k\ell}(\tau-\xi)\,\mathrm{d}\xi \right) = T_k^{\text{const}} + \sum_{\ell=1}^{|\Theta|} M_{k,\ell}\left(S_\ell^*\right)$$

with the number of nodes n of the thermal RC circuit, $T_k^{\text{const}} = T_k^{\text{init}} + \sum_{\ell=1}^{n} \left(u_\ell^{\text{idle}} \cdot \int_0^\tau H_{k\ell}(t-\xi)\,\mathrm{d}\xi \right)$ and $M_{k,\ell}\left(S_\ell^*\right) = \rho_{\ell\ell} \cdot f_\ell^3 \cdot \int_0^\tau S_\ell^*(\xi) \cdot \widetilde{H}_{k\ell}(\tau-\xi)\,\mathrm{d}\xi$. In the last step, we used the fact, that $M_{k,\ell}$ is all zero if node ℓ does not correspond to a processing component, to change the bound of summation. T_k^{const} is independent of the assignment, thus it is calculated once for all possible task assignments. Suppose that the worst-case chip temperature is calculated for two task assignments that only differ in the assignment of task ν_y. In particular, the first assignment maps task ν_y to component i and the second assignment maps task ν_y to component j. After calculating the worst-case chip temperature for the first assignment, the only elements that have to be recalculated for the second task assignment are all $M_{*,i}$ and $M_{*,j}$. In particular, the number of elements to be recalculated is reduced by a factor of $\frac{|\Theta|-2}{|\Theta|}$.

6 Case Studies

In order to study the viability of the proposed approaches, we solved TMP with four different solvers for different task sets and floorplans. To this end, the discussed methods are implemented in the real-time calculus toolbox [19].

6.1 System Description

We are targeting a reconfigurable homogeneous multi-core ARM platform with a variable number of processing components. The maximum frequency of all

Table 1. Thermal configuration of HotSpot and the power dissipation parameters of the power model defined as in (10)

(a) Thermal configuration of HotSpot.

Parameter	Symbol	Value
Silicon thermal cond. [W/(m · K)]	k_{chip}	150
Silicon specific heat [J/(m^3 · K)]	p_{chip}	$1.75 \cdot 10^6$
Thickness of the chip [mm]	t_{chip}	3.5
Convection resistance [K/W]	r_{convec}	2
Heatsink thickness [mm]	t_{sink}	0.01
Heatsink thermal cond. [W/(m · K)]	k_{sink}	400
Heatsink specific heat [J/(m^3 · K)]	p_{sink}	$3.55 \cdot 10^6$
Ambient temperature [K]	T_{amb}	300

(b) Power configuration.

Parameter	Value
$\phi_{\ell\ell}$ [W/K]	0.0228
$\rho_{\ell\ell}$ [W/GHz^3]	3.936
ψ_ℓ [W]	−2.756

cores is 1.6 GHz and an EDF scheduler is running on each core to arbitrate between events of different tasks assigned to the same core. HotSpot [16] is used to calculate the thermal parameters of the platform, i.e., **C**, **G**, and **K** matrices, see Table 1(a) for the detailed thermal configuration. The temperature-dependency of leakage power is addressed by linearizing the model described in [15] and the parameters of the power model for the platform with a 3×1 core layout are summarized in Table 1(b). As we consider a homogeneous platform, every component has the same power values. In all experiments, the traces start from the steady-state temperature in 'idle' mode, i.e., $\mathbf{T}^0 = (\mathbf{T}^\infty)^i$.

6.2 Performance of Four Different TMP Solvers

The optimal solution of the TMP can exhaustively be calculated for small task sets and platforms with a low number of processing components. We assess the performance of metaheurstics compared to exhaustive search for three different platforms and six different sets of tasks. The considered hardware platforms have a 3×1, 2×2, and 3×2 layout with three, four, and six cores, respectively. The event model of task ν_j is described using a set of two parameters, namely the period p_{ν_j} and jitter j_{ν_j} [9]. In particular, the period p_{ν_j} is uniformly chosen from $[1, 400\,]ms$, its jitter j_{ν_j} is uniformly chosen from $[1\,\text{ms}, 2 \cdot p_{\nu_j}]$, and the computational demand is uniformly chosen from $\left[1, p_{\nu_j} \cdot f^{\max}/5\right]$ cycles with $f^{\max} = 1.6\,\text{GHz}$. The real-time deadline of a task is equal to its period. Finally, the number of tasks in one set is randomly chosen between four and six tasks.

In total, we evaluate the performance of four different solvers for TMP. The first one exhaustively tests all possible assignments to compute the optimal solution from the TMP formulation. The optimal solution is compared, on the one hand, to the solution of simulated annealing [20] and, on the other hand, to the solution of a local search algorithm, and the average peak temperature of 20 feasible random assignments. The local search algorithm fully exploits the characteristics of the considered peak temperature computation algorithm as described in Section 5.3

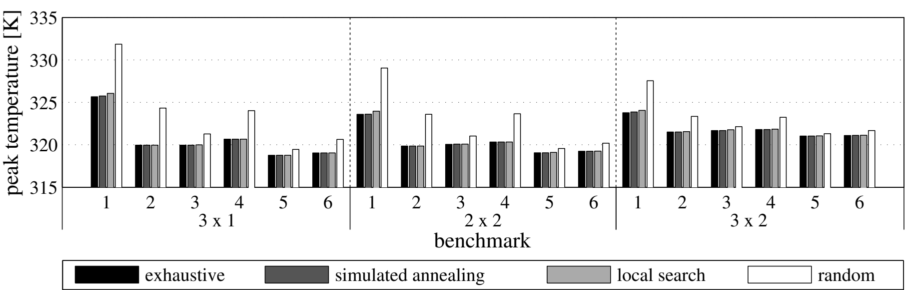


Fig. 8. Performance of four different TMP solvers. The hardware platforms have a 3×1, 2×2, and 3×2 layout, respectively.

by testing the peak temperature of all neighbor assignments, and then, selecting the assignment that minimizes the peak temperature the most.

Fig. 8 compares the performance of the four different solvers. The peak temperature of the optimal task assignment is on average 2.27 K lower than the peak temperature of the random assignments. Simulated annealing found in all considered benchmarks an assignment that has a peak temperature that is no more than 0.11 K higher than the optimal assignment. This shows that probabilistic metaheuristics are well suited to solve TMP. The local search algorithm calculates an assignment that is no more than 0.4 K higher than the peak temperature of the optimal assignment. In particular, one can see that there are a few task sets where the local search algorithm proposes a task assignment that is significantly worse than the optimal solution. This could be prevented by extending the local search algorithm such that it does not only consider the direct neighborhood of the current assignment, but all assignments up to its k-th neighborhood. The difference in terms of peak temperature between the solvers becomes even larger if the number of tasks per task set is increased as more local optima emerge. As frequency reduction damps the effect of burst on the peak temperature, most solvers are able to find an acceptable solution. However, once the damping is removed, the peak temperature might drastically increase, which in turn results in higher peak temperature differences between the solvers.

On a 2.55 GHz Intel Core i5-2400S processor, calculating the optimal solution for the hardware platform with 6 cores took on average 1.52 h. Simulated annealing and the local search algorithm finished on average in 22.6 s and 0.77 s, respectively. Finally, calculating the peak temperature of 20 random assignments took on average 3.1 s.

6.3 Performance for Different Utilizations and Floorplans

In the second case study, we evaluate the worst-case chip temperature for different floorplans and utilizations. The layout of the considered platforms is 3×1, 3×2, and 4×4 with 3, 6, and 16 cores, respectively. In all benchmarks, the TMP is solved by simulated annealing. The task sets are iteratively generated, starting with an initial size of $|\Theta|$ randomly generated tasks. Then, as long as the

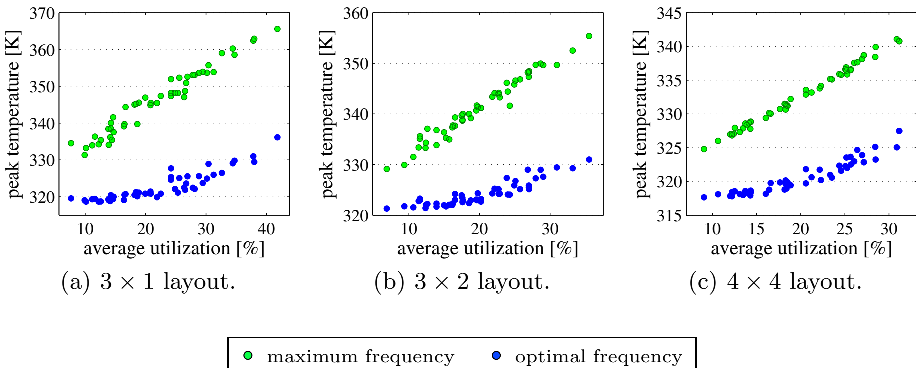


Fig. 9. Worst-case chip temperature for three hardware platforms. To calculate the worst-case peak temperature, TMP is once solved under the assumption that all processing components are running at maximum frequency, and once under the assumption that the components are running at optimal frequency.

system is schedulable, we add a new randomly generated task to the collection. In total, we generate 50 different task sets for each hardware platform.

For each benchmark, we resolve the TMP once under the assumption that all processing components are running at maximum frequency, i.e., 1.6 GHz, and once under the assumption that the components are running at their optimal frequency such that each benchmark is characterized by a triple $(T^*_{f_{\max}}, T^*_{f_{\text{opt}}}, util)$. $T^*_{f_{\max}}$ is the peak temperature when the components are running at maximum frequency, $T^*_{f_{\text{opt}}}$ is the peak temperature when the components are running at optimal frequency and *util* is the average utilization of all cores when the components are running at their optimal frequency and the jitter is ignored. Even thought the components are running at their optimal frequency, the utilization is not 100 % as the jitter has a high impact on the selection of the frequencies.

Finally, in Fig. 9, we plot $T^*_{f_{\max}}$ and $T^*_{f_{\text{opt}}}$ as a function of *util* for three hardware platforms. It shows that the chip temperature can drastically be reduced when the processing components are running at their optimal frequency. In particular, the peak temperature can be reduced on average by 23.6 K for the 3 × 1 layout, by 17.0 K for the 3 × 2 layout, and by 12.1 K for the 4 × 4 layout. Furthermore, Fig. 9 shows that the worst-case chip temperature does not necessarily increase with the utilization as different amounts of non-determinism might cause higher chip temperatures for lower utilizations.

7 Related Work

Xie and Hung [21] were the first to identify the topic of thermal-aware task allocation and scheduling. Later, a convex optimization technique for temperature-aware frequency assignment is proposed to maximize the performance under temperature constraints [5] and the task scheduling problem is statically solved using integer linear programming for minimizing energy, and reducing hot spots [7].

The minimization of the peak temperature in the presence of real-time deadlines is formulated as a nonlinear programming problem in [22]. A mixed-integer linear programming formulation for assigning and scheduling tasks with hard real-time constraints to reduce the peak temperature is proposed in [4]. Finally, Fisher et al. [6] proposed a global scheduling algorithm such that all cores are running at their ideally preferred speed, and the peak temperature is minimized. However, as the peak temperature is calculated in these works by either steady-state temperature analysis or transient temperature evolution, the proposed methods cannot be used to optimize the task to processing component assignment of a system with non-deterministic workload and hard real-time guarantees. As high chip temperatures can significantly reduce the system's performance, real-time constraints can only be guaranteed if the worst-case chip temperature is incorporated in real-time analysis, at design-time.

HotSpot [16] is the most popular simulator for thermal analysis. However, as thermal simulation methods only cover a fraction of all possible system behaviors, they are not able to capture the maximum temperature of an application with non-deterministic workload. Tackling this challenge, a method to calculate the worst-case chip temperature of a multi-core system with non-deterministic workload has been proposed in [17]. In comparison with the method proposed in this paper, the authors of [17] use periodic event streams with burst [9] for the event model of every processing component.

8 Conclusion

In this paper, we formulated the thermal-aware task assignment and frequency selection problem to optimize the worst-case chip temperature under real-time constraints as a nonlinear binary integer problem. In order to solve the proposed problem, we described a novel analytical method to calculate an upper bound on the maximum chip temperature under all feasible scenarios of task arrivals. Each core has its own clock domain and the static assigned frequencies correspond to the minimum operation frequencies such that no real-time deadline is missed. The considered thermal model is able to address various thermal effects like heat exchange between neighboring cores and temperature-dependent leakage power. Arrival curves from real-time calculus are used to upper bound the task's workload in any time interval. Case studies have shown that the worst-case chip temperature of an embedded multi-core system can be reduced by more than 20 K by assigning each processing component its ideally preferred frequency and selecting the optimal task to processing component assignment.

Acknowledgments. This work was supported by EU FP7 projects EURETILE and PRO3D, under grant numbers 247846 and 249776.

References

1. Gunther, S., Binns, F., Carmean, D., Hall, J.: Managing the Impact of Increasing Microprocessor Power Consumption. Intel Technology Journal 5(1), 1–9 (2001)

2. Donald, J., Martonosi, M.: Techniques for Multicore Thermal Management: Classification and New Exploration. In: Proc. Int'l Symposium on Computer Architecture, ISCA, Boston, MA, USA, pp. 78–88. IEEE (2006)
3. Isci, C., Buyuktosunoglu, A., Cher, C.Y., Bose, P., Martonosi, M.: An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In: Proc. Int'l Symposium on Microarchitecture, MICRO, pp. 347–358. IEEE (2006)
4. Chantem, T., Dick, R., Hu, X.: Temperature-Aware Scheduling and Assignment for Hard Real-Time Applications on MPSoCs. In: Proc. Design, Automation and Test in Europe, DATE, Munich, Germany, pp. 288–293. ACM/IEEE (2008)
5. Murali, S., Mutapcic, A., Atienza, D., Gupta, R., Boyd, S., De Micheli, G.: Temperature-Aware Processor Frequency Assignment for MPSoCs Using Convex Optimization. In: Proc. Int'l Conf. on Hardware/Software Codesign and System Synthesis, CODES+ISSS, Salzburg, Austria, pp. 111–116. ACM (2007)
6. Fisher, N., Chen, J.J., Wang, S., Thiele, L.: Thermal-Aware Global Real-Time Scheduling on Multicore Systems. In: Proc. Real-Time and Embedded Technology and Applications Symposium, RTAS, San Francisco, USA, pp. 131–140. IEEE (2009)
7. Coskun, A., Rosing, T., Whisnant, K., Gross, K.: Static and Dynamic Temperature-Aware Scheduling for Multiprocessor SoCs. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 16(9), 1127–1140 (2008)
8. Thiele, L., Chakraborty, S., Naedele, M.: Real-Time Calculus for Scheduling Hard Real-Time Systems. In: Proc. Int. Symposium on Circuits and Systems, ISCAS, Geneva, Switzerland, vol. 4, pp. 101–104. IEEE (2000)
9. Henia, R., Hamann, A., Jersak, M., Racu, R., Richter, K., Ernst, R.: System Level Performance Analysis - The SymTA/S Approach. IEEE Proc. Comp. and Digital Tech. 152(2), 148–166 (2005)
10. Baruah, S., Mok, A., Rosier, L.: Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor. In: Proc. Real-Time Systems Symposium, RTSS, Lake Buena Vista, FL, USA, pp. 182–190. IEEE (1990)
11. Wandeler, E., Maxiaguine, A., Thiele, L.: Performance Analysis of Greedy Shapers in Real-Time Systems. In: Proc. Design, Automation and Test in Europe, DATE, Munich, Germany, pp. 444–449 (2006)
12. Chen, J.J., Wang, S., Thiele, L.: Proactive Speed Scheduling for Real-Time Tasks under Thermal Constraints. In: Proc. Real-Time and Embedded Technology and Applications Symposium, RTAS, San Francisco, CA, USA, pp. 141–150. IEEE (2009)
13. Rabaey, J.M., Chandrakasan, A., Nikolic, B.: Digital Integrated Circuits, 3rd edn. Prentice Hall Press (2008)
14. Liu, Y., Dick, R.P., Shang, L., Yang, H.: Accurate Temperature-Dependent Integrated Circuit Leakage Power Estimation is Easy. In: Proc. Design, Automation and Test in Europe, DATE, Nice, France, pp. 1526–1531 (2007)
15. Skadron, K., et al.: Temperature-Aware Microarchitecture: Modeling and Implementation. ACM Trans. Architec. Code Optim. 1(1), 94–125 (2004)
16. Huang, W., Ghosh, S., Velusamy, S., Sankaranarayanan, K., Skadron, K., Stan, M.: HotSpot: A Compact Thermal Modeling Methodology for Early-Stage VLSI Design. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 14(5), 501–513 (2006)
17. Schor, L., Bacivarov, I., Yang, H., Thiele, L.: Worst-Case Temperature Guarantees for Real-Time Applications on Multi-Core Systems. In: Proc. Real-Time and Embedded Technology and Applications Symposium, RTAS, Beijing, China, pp. 87–96. IEEE (2012)

18. Ferreira, P.: Sorting Continuous-Time Signals: Analog Median and Median-Type Filters. IEEE Trans. Signal. Proces. 49(11), 2734–2744 (2001)
19. Wandeler, E., Thiele, L.: Real-Time Calculus (RTC) Toolbox (2006), `http://www.mpa.ethz.ch/Rtctoolbox`
20. Kirkpatrick, S., Gelatt, C., Vecchi, M.: Optimization by Simulated Annealing. Science 220(4598), 671–680 (1983)
21. Xie, Y., Hung, W.L.: Temperature-Aware Task Allocation and Scheduling for Embedded Multiprocessor Systems-on-Chip (MPSoC) Design. The Journal of VLSI Signal Processing 45(3), 177–189 (2006)
22. Liu, Y., Yang, H., Dick, R., Wang, H., Shang, L.: Thermal vs Energy Optimization for DVFS-Enabled Processors in Embedded Systems. In: Proc. Int'l Symposium on Quality Electronic Design, ISQED, San Jose, CA, USA, pp. 204–209. IEEE (2007)

Appendix: Proof of Theorem 2

In the following, we will show that $T^*_{k,\ell}(\tau) \geq T_{k,\ell}(\tau)$ for any valid $T_{k,\ell}(\tau)$. Rewriting (14) with (15) leads to $T_{k,\ell}(t) = u_\ell^{\text{idle}} \cdot \int_0^t H_{k\ell}(t-\xi)\,\mathrm{d}\xi + \rho_{\ell\ell} \cdot f_\ell^3 \cdot \int_0^t S_\ell(\xi) \cdot H_{k\ell}(t-\xi)\,\mathrm{d}\xi$. Then we have:

$$T^*_{k,\ell}(\tau) - T_{k,\ell}(\tau) = \rho_{\ell\ell} \cdot f_\ell^3 \cdot \left(\int_0^\tau S^*_\ell(\xi) \cdot \widetilde{H}_{k\ell}(\tau-\xi)\,\mathrm{d}\xi - \int_0^\tau S_\ell(\xi) \cdot H_{k\ell}(\tau-\xi)\,\mathrm{d}\xi \right)$$

with $\rho_{\ell\ell} \cdot f_\ell^3 > 0$. In other words, we have to show that $\int_0^\tau S^*_\ell(\xi) \cdot \widetilde{H}_{k\ell}(\tau-\xi)\,\mathrm{d}\xi \geq \int_0^\tau S_\ell(\xi) \cdot H_{k\ell}(\tau-\xi)\,\mathrm{d}\xi$.

Discretization

In order to simplify the proof technicalities, we suppose discrete time, i.e., $S_\ell(t), S^*_\ell(t), \widetilde{H}_{k\ell}(t)$, and $H_{k\ell}(t)$ may change values only at multiples of δ and are constant for $t \in [r \cdot \delta, (r+1) \cdot \delta)$ for all $r \geq 0$. With $r_\tau = \tau \cdot \delta$, we have:

$$\int_0^\tau S^*_\ell(\xi) \cdot \widetilde{H}_{k\ell}(\tau-\xi)\,\mathrm{d}\xi = \delta \cdot \sum_{r=0}^{r_\tau-1} S^*_\ell[r] \cdot \widetilde{H}_{k\ell}[r_\tau - 1 - r] \tag{25}$$

and

$$\int_0^\tau S_\ell(\xi) \cdot H_{k\ell}(\tau-\xi)\,\mathrm{d}\xi = \delta \cdot \sum_{r=0}^{r_\tau-1} S_\ell[r] \cdot H_{k\ell}[r_\tau - 1 - r] \quad . \tag{26}$$

Next, we show that $\sum_{r=0}^{r_\tau-1} S_\ell[r] \cdot H_{k\ell}[r_\tau-1-r] \leq \sum_{r=0}^{r_\tau-1} S^*_\ell[r] \cdot \widetilde{H}_{k\ell}[r_\tau-1-r]$ for all S_ℓ that satisfy (6), by induction. To this end, we will prove that:

$$\underbrace{\sum_{r=w}^{w+\pi-1} S_\ell[r] \cdot H_{k\ell}[r_\tau-1-r]}_{\mathcal{T}(\pi,w,S_\ell)} \leq \underbrace{\sum_{r=r_\tau-\pi}^{r_\tau-1} S^*_\ell[r] \cdot \widetilde{H}_{k\ell}[r_\tau-1-r]}_{\mathcal{T}^*(\pi)} \tag{27}$$

for any $\pi \in [0, r_\tau]$ and any $w \in [0, r_\tau - \pi]$.

Base Case

First, we show that the statement is true for $\pi = 1$. Rewriting (27) with $\pi = 1$ leads to $S_\ell[w] \cdot H_{k\ell}[r_\tau - 1 - w] \leq S^*_\ell[r_\tau - 1] \cdot \widetilde{H}_{k\ell}[r_\tau - 1 - (r_\tau - 1)] = S^*_\ell[r_\tau - 1] \cdot \widetilde{H}_{k\ell}[0]$. As $S^*_\ell[r_\tau - 1] = 1$ and $\widetilde{H}_{k\ell}[0] \geq H_{k\ell}[\eta]$ for all $\eta \geq 0$, the statement is true for $\pi = 1$.

Induction Hypothesis

Next, we show that the statement is true for π if it is true for $\pi - 1$. In other words, we assume as *induction hypothesis* that:

$$\mathcal{T}(\pi - 1, w, S_\ell) \leq \mathcal{T}^*(\pi - 1) \tag{28}$$

holds for all $w \in [0, r_\tau - \pi + 1]$.

Induction Step

Let us prove by contradiction that (27) is true for any π. Therefore, assume for contradiction that there exists a $\overline{w}$ such that:

$$\mathcal{T}(\pi, \overline{w}, S_\ell) > \mathcal{T}^*(\pi) \ . \tag{29}$$

Now, we differ between the following cases:

Case 1: $S_\ell[\overline{w} + \pi - 1] = 0$.
The contradiction follows from $\mathcal{T}(\pi, \overline{w}, S_\ell) = \mathcal{T}(\pi - 1, \overline{w}, S_\ell) + S_\ell[\overline{w} + \pi - 1] \cdot H_{k\ell}[r_\tau - 1 - (\overline{w} + \pi - 1)] = \mathcal{T}(\pi - 1, \overline{w}, S_\ell) + 0 \cdot H_{k\ell}[r_\tau - 1 - (\overline{w} + \pi - 1)] \leq \mathcal{T}^*(\pi - 1) \leq \mathcal{T}^*(\pi)$.

Case 2: $S_\ell[\overline{w} + \pi - 1] = 1,\ S^*_\ell[r_\tau - \pi] = 1$.
As $\mathcal{T}^*(\pi) = \mathcal{T}^*(\pi - 1) + \widetilde{H}_{k\ell}[r_\tau - 1 - (r_\tau - \pi)]$ and $\mathcal{T}(\pi, \overline{w}, S_\ell) = \mathcal{T}(\pi - 1, \overline{w}, S_\ell) + H_{k\ell}[r_\tau - 1 - (\overline{w} + \pi - 1)]$, it follows that $\widetilde{H}_{k\ell}[\pi - 1] < H_{k\ell}[r_\tau - (\overline{w} + \pi)]$.
First, we show that $\widetilde{H}_{k\ell}[\pi - 1] < H_{k\ell}[r_\tau - (\overline{w} + \pi)]$ implies that $H_{k\ell}[r_\tau - \overline{w}] \leq \widetilde{H}_{k\ell}[\pi - 1]$. As $H_{k\ell}$ is a non-negative unimodal function, the condition $H_{k\ell}[r_\tau - \overline{w}] > \widetilde{H}_{k\ell}[\pi - 1]$ requires that all $\pi + 1$ elements $H_{k\ell}[\eta]$ for $\eta \in [r_\tau - (\overline{w} + \pi), r_\tau - \overline{w}]$ fulfill $H_{k\ell}[\eta] > \widetilde{H}_{k\ell}[\pi - 1]$, see Fig. 10 for an illustration. However, as $\widetilde{H}_{k\ell}[\pi - 1]$ is the π-th largest element of $H_{k\ell}$, this is a contradiction, and $H_{k\ell}[r_\tau - \overline{w}] \leq \widetilde{H}_{k\ell}[\pi - 1]$.
As $\mathcal{T}(\pi - 1, w, S_\ell) \leq \mathcal{T}^*(\pi - 1)$ for any w, in particular also for $w = \overline{w} + 1$, we find $\mathcal{T}(\pi, \overline{w}, S_\ell) \leq H_{k\ell}[r_\tau - \overline{w}] + \mathcal{T}(\pi - 1, \overline{w} + 1, S_\ell) \leq \widetilde{H}_{k\ell}[\pi - 1] + \mathcal{T}^*(\pi - 1) = \mathcal{T}^*(\pi)$, which is a contradiction.

Case 3: $S_\ell[\overline{w} + \pi - 1] = 1,\ S^*_\ell[r_\tau - \pi] = 0$.
From $S^*_\ell[r_\tau - \pi] = 0$ follows that $\sum_{r=r_\tau-\pi+1}^{r_\tau-1} S^*_\ell[r] = \gamma[\pi - 2] = \sum_{r=r_\tau-\pi}^{r_\tau-1} S^*_\ell[r] = \gamma[\pi - 1]$.

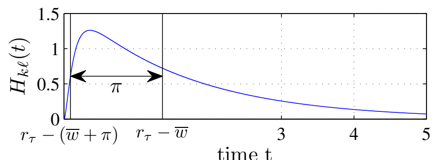


Fig. 10. Sketch of the proof that in Case 2, $\widetilde{H}_{k\ell}[\pi - 1] < H_{k\ell}[r_\tau - (\overline{w} + \pi)]$ implies that $\widetilde{H}_{k\ell}[\pi - 1] \geq H_{k\ell}[r_\tau - \overline{w}]$

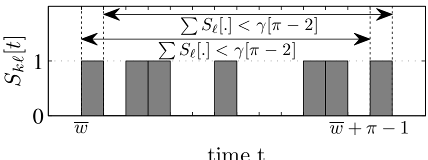


Fig. 11. Sketch of Case 3(b) that illustrates how the accumulated computing time is upper bounded

a) $S_\ell[\overline{w}] = 0$.
From $S_\ell[\overline{w}] = 0$ follows that $\mathcal{T}(\pi, \overline{w}, S_\ell) = \mathcal{T}(\pi - 1, \overline{w} + 1, S_\ell) \leq \mathcal{T}^*(\pi - 1) = \mathcal{T}^*(\pi)$, which is a contradiction.

b) $S_\ell[\overline{w}] = 1$.
First note that $\sum_{r=\overline{w}}^{\overline{w}+\pi-1} S_\ell[r] \leq \gamma_\ell[\pi - 1] = \gamma_\ell[\pi - 2]$. As $S_\ell[\overline{w}] = 1$, we know that $\sum_{r=\overline{w}+1}^{\overline{w}+\pi-1} S_\ell[r] < \sum_{r=r_\tau-\pi+1}^{r_\tau-1} S_\ell^*[r] = \gamma_\ell[\pi - 2]$ and as $S_\ell[\overline{w} + \pi - 1] = 1$, we know that $\sum_{r=\overline{w}}^{\overline{w}+\pi-2} S_\ell[r] < \sum_{r=r_\tau-\pi+1}^{r_\tau-1} S_\ell^*[r] = \gamma_\ell[\pi - 2]$, see also Fig. 11.
In case that $H_{k\ell}[r_\tau - 1 - \overline{w}] < H_{k\ell}[r_\tau - 1 - (\overline{w} + \pi - 1)]$, we know that $H_{k\ell}[\eta] \geq H_{k\ell}[r_\tau - 1 - \overline{w}]$ for any $\eta \in [r_\tau - (\overline{w} + \pi), r_\tau - \overline{w} - 1]$ (see Fig. 10). Therefore, there exists:

$$\overline{S}_\ell[r] = \begin{cases} 0 & r = \overline{w} \\ 1 & r = \overline{w}' \\ S_\ell[r] & \text{otherwise} \end{cases} \tag{30}$$

with $\overline{w} < \overline{w}' < \overline{w} + \pi - 1$ and $S_\ell[\overline{w}'] = 0$. As $H_{k\ell}[r_\tau - 1 - \overline{w}'] \geq H_{k\ell}[r_\tau - 1 - \overline{w}]$, we have $\mathcal{T}(\pi, \overline{w}, S_\ell) \leq \mathcal{T}(\pi, \overline{w}, \overline{S}_\ell)$. Similarly, we can find a $\overline{S}_\ell$ and $\overline{w}'$ for the case $H_{k\ell}[r_\tau - 1 - \overline{w}] \geq H_{k\ell}[r_\tau - 1 - (\overline{w} + \pi - 1)]$.
Now, applying Case 1 or Case 3.a to $\overline{S}_\ell$ shows that $\mathcal{T}(\pi, \overline{w}, \overline{S}_\ell) \leq \mathcal{T}^*(\pi)$, and therefore, $\mathcal{T}(\pi, \overline{w}, S_\ell) \leq \mathcal{T}(\pi, \overline{w}, \overline{S}_\ell) \leq \mathcal{T}^*(\pi)$, which is the contradiction.

As we have shown that (27) is true for any π, it is particularly true for $\pi = r_\tau$, and the theorem follows. □

Component Assemblies in the Context of Manycore*

Ananda Basu[1], Saddek Bensalem[1], Marius Bozga[1], Paraskevas Bourgos[1], Mayur Maheshwari[1], and Joseph Sifakis[1,2]

[1] UJF-Grenoble 1 / CNRS, VERIMAG UMR 5104, Grenoble, F-38041, France
{basu,bensalem,bozga,bourgos,maheshwari,sifakis}@imag.fr
[2] RISD Laboratory, EPFL
joseph.sifakis@epfl.ch

Abstract. We present a component-based software design flow for building parallel applications running on top of manycore platforms. The flow is based on the BIP - Behaviour, Interaction, Priority - component framework and its associated toolbox. It provides full support for modeling of application software, validation of its functional correctness, modeling and performance analysis on system-level models, code generation and deployment on target manycore platforms. The paper details some of the steps of the design flow. The design flow is illustrated through the modeling and deployment of two applications, the Cholesky factorization and the MJPEG decoding on MPARM, an ARM-based manycore platform. We emphasize the merits of the design flow, notably fast performance analysis as well as code generation and efficient deployment on manycore platforms.

1 Introduction

The emergence of manycore platforms is nowadays challenging the design practices for embedded software. Manycore platforms built on increasingly complex 2D or 3D hardware architectures which, besides a high number of computational cores, usually include complex memory/cache hierarchies, synchronization patterns and/or communication buses and networks. Commonly, all hardware resources are either partially or fully exposed to software developers. By doing so, one expects optimized exploitation of resources while meeting requirements for both software performance (e.g., real-time requirements) and efficient platform management (e.g., thermal and power efficiency).

Concurrency is paramount for boosting software performance on manycore platforms. Nonetheless, correct and fast development of highly parallel, fine-grain concurrent software is known to be notoriously hard even for expert developers. In general, the inherent complexity of concurrent (handwritten) software

* The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement no. 248776 (PRO3D) and from ARTEMIS JU grant agreement ARTEMIS-2009-1-100230 (SMECY).

B. Beckert et al. (Eds.): FMCO 2011, LNCS 7542, pp. 314–333, 2013.
© Springer-Verlag Berlin Heidelberg 2013

is hardly manageable by current verification and validation methods and tools. Moreover, software adaptation and deployment to selected manycore platform targets usually require significant manual transformation, with no strong guarantees about their correctness.

The PRO3D project [32] proposes a holistic approach for the development of embedded applications on top of manycore 3D platforms. PRO3D activities range from programming to architecture exploration and fabrication technologies. The major challenges are the thermal management of 3D platforms and the rigorous, tool-supported design flow of parallel application software.

We propose and implement a design flow for applications based on the BIP component framework [6]. This flow sticks to the general principles of *rigorous design* introduced in [8]. It has several key features, namely:

- it is *model-based*, that is, both application software and mixed hardware/ software system descriptions are modeled by using a single, semantic framework. As stated in [8], this allows maintaining the coherency along with the flow by proving that various transformations used to move from one description to another preserve essential properties.
- it is *component-based*, that is, it provides primitives for building composite components as the composition of simpler components. Using components reduces development time by favoring component reuse and provides support for incremental analysis and design, as introduced in [10,11,13]
- it is *tool-supported*, that is, all steps in the design flow are realized automatically by tools. This ensures significant productivity gains, in particular due to elimination of potential errors that can occur in manual transformations.

To the best of our knowledge, the BIP design flow is unique as it uses a single semantic framework to support application modeling, validation of functional correctness, performance analysis on system models and code generation for manycore platforms. Building faithful system models is mandatory for validation and performance analysis of concurrent software running on manycore platforms. Existing system modeling formalisms either seek generality at the detriment of rigorousness, such as SySML [30] and AADL [20] or have a limited scope as they are based on specific models of computation such as Ptolemy [18]. Simulation based methods use ad-hoc executable system models such as [22] or tools based on SystemC [28]. The latter provide cycle-accurate results, but in general, they have long simulation time as a major drawback. As such, these tools are not adequate for thorough exploration of hardware platform dynamics, neither for estimating effects on real-life software execution. Alternatives include trace-based co-simulation methods as used in Spade [26], Sesame [19] or Daedalus [29]. Additionally, there exist much faster techniques that work on abstract system models e.g., Real Time Calculus [36] and SymTA/S [4]. They use formal analytical models representing a system as a network of nodes exchanging streams. They often oversimplify the dynamics of the execution characterized by execution times. Moreover, they allow only estimation of pessimistic worst-case quantities (delays, buffer sizes, etc) and require adequate abstract models of the

application software. Building such models entails an additional significant modeling effort. Similar difficulties arise in performance analysis techniques based on Timed-Automata [3,33]. These can be used for modeling and solving scheduling problems. An approach combining simulation and analytic models is presented in [23], where simulation results can be propagated to analytic models and vice versa through adequate interfaces.

The paper is organized as follows. Section 2 provides a brief overview of the BIP component framework and toolbox. Section 3 introduces the BIP design flow and details several of its steps. An illustration of the design flow is provided in section 4. We provide results about performance analysis and implementation of two non-trivial concurrent applications on manycore. Finally, section 5 concludes and provides future work directions.

2 The BIP Framework

The BIP – Behaviour / Interaction / Priority – framework [6] is aiming at design and analysis of complex, heterogeneous embedded applications. BIP is a highly expressive, component-based framework with rigorous semantical basis. It allows the construction of complex, hierarchically structured models from atomic components characterized by their behavior and their interfaces. Such components are transition systems enriched with data. Transitions are used to move from a source to a destination location. Each time a transition is taken, component data (variables) may be assigned new values, computed by user-defined functions (in C). Atomic components are composed by layered application of interactions and priorities. Interactions express synchronization constraints and define the transfer of data between the interacting components. Priorities are used to filter amongst possible interactions and to steer system evolution so as to meet performance requirements e.g., to express scheduling policies.

Atomic Components. We define *atomic components* as transition systems extended with a set of ports and a set of variables. Formally, an *atomic component* B is a labelled transition system represented by a tuple (Q, X, P, T) where Q is a set of *control locations*, X is a set of *variables*, P is a set of *communication ports* and T is a set of *transitions*. Each transition τ is of the form (q, p, g, f, q') where $q, q' \in Q$ are control locations, $p \in P$ is a port, g is the *guard* and f is the *update function* of τ. g is a predicate defined over variables in X and f is a function (or a sequential procedure) that computes new values for X according to the current ones.

Interactions. In order to compose a set of n atomic components $\{B_i = (Q_i, X_i, P_i, T_i)\}_{i=1}^n$, we assume that their respective sets of ports and variables are pairwise disjoint; i.e., for all $i \neq j$ we require that $P_i \cap P_j = \emptyset$ and $X_i \cap X_j = \emptyset$. We define the global set $P \stackrel{def}{=} \bigcup_{i=1}^n P_i$ of ports. An *interaction* a is a triple (P_a, G_a, F_a), where $P_a \subseteq P$ is a set of ports, G_a is a guard, and F_a is a data transfer function. By definition P_a contains at most one port from each component. We denote $P_a = \{p_i\}_{i \in I}$ with $I \subseteq \{1..n\}$ and $p_i \in P_i$. We assume that G_a

and F_a are defined on the variables of participating components, (i.e. $\bigcup_{i\in I} X_i$). We denote by F_a^i the restriction of F_a on variables X_i.

Priorities. Given a set γ of interactions, we define a priority as a strict partial order $\pi \subseteq \gamma \times \gamma$. We write $a\pi b$ for $(a,b) \in \pi$, to express the fact that interaction a has lower priority than interaction b.

Composite Components. A *composite component* $\pi\gamma(B_1,\ldots,B_n)$ is defined by a set of atomic components $B_1,\cdots,B_n$, composed by a set of interactions γ and a priority $\pi \subseteq \gamma \times \gamma$. If π is the empty relation, then we may omit π and simply write $\gamma(B_1,\cdots,B_n)$.

A global state of $\pi\gamma(B_1,\cdots,B_n)$ where $B_i = (Q_i, X_i, P_i, T_i)$ is defined by a couple (q,v), where $q = (q_1,\cdots,q_n)$ is a tuple of control locations such that $q_i \in Q_i$ and $v = (v_1,\cdots,v_n)$ is a tuple of valuations of variables such that $v_i \in Val(X_i) = \{\sigma : X_i \rightarrow \mathcal{D}\}$, for all $i = 1,\cdots n$ and for $\mathcal{D}$ being some universal data domain. The behavior of a composite component without priority $\gamma(B_1,\cdots,B_n)$ is a labeled transition system $(S,\gamma,\rightarrow_\gamma)$, where $S = \bigotimes_{i=1}^n Q_i \times \bigotimes_{i=1}^n Val(X_i)$ and $\rightarrow_\gamma$ is the least set of transitions satisfying the rule:

$$\frac{\begin{array}{c} a = (\{p_i\}_{i\in I}, G_a, F_a) \in \gamma \quad v_a = \{v_i\}_{i\in I} \quad G_a(v_a) \\ \forall i \in I.\ (q_i, g_i, p_i, f_i, q_i') \in T_i \quad g_i(v_i) \quad v_i' = f_i(F_a^i(v_a)) \\ \forall i \notin I.\ (q_i, v_i) = (q_i', v_i') \end{array}}{((q_1,\ldots,q_n),(v_1,\ldots,v_n)) \xrightarrow{a}_\gamma ((q_1',\ldots,q_n'),(v_1',\ldots,v_n'))} \ [\textsc{interaction}]$$

Intuitively, the inference rule INTERACTION specifies that a composite component $B = \gamma(B_1,\ldots,B_n)$ can execute an interaction $a \in \gamma$, iff (1) for each port $p_i \in P_a$, the corresponding atomic component B_i allows a transition from the current location labelled by p_i (i.e. the corresponding guard g_i evaluates to true), and (2) the guard G_a of the interaction evaluates to true. If the two above conditions hold for an interaction a at state (q,v), a is *enabled* at that state. Execution of a modifies participating components' variables by first applying the data transfer function F_a on variables of all interacting components and then the update function f_i for each interacting component. The (local) states of components that do not participate in the interaction stay unchanged.

We define the behavior of the composite component $B = \pi\gamma(B_1,\ldots,B_n)$ as the labeled transition system $(S,\gamma,\rightarrow_{\pi\gamma})$ where $\rightarrow_{\pi\gamma}$ is the least set of transitions satisfying the rule:

$$\frac{(q,v) \xrightarrow{a}_\gamma (q',v') \qquad \forall a' \in \gamma.\ a\pi a' \Longrightarrow (q,v) \stackrel{a'}{\not\rightarrow}_\gamma}{(q,v) \xrightarrow{a}_{\pi\gamma} (q',v')} \ [\textsc{priority}]$$

The inference rule PRIORITY filters out interactions which are not maximal with respect to the priority order. An interaction is executed only if no other one with higher priority is enabled.

Example 1. Figure 1 shows a graphical representation of an example model in BIP. It consists of atomic components *Sender*, *Buffer* and *Receiver*. The behavior

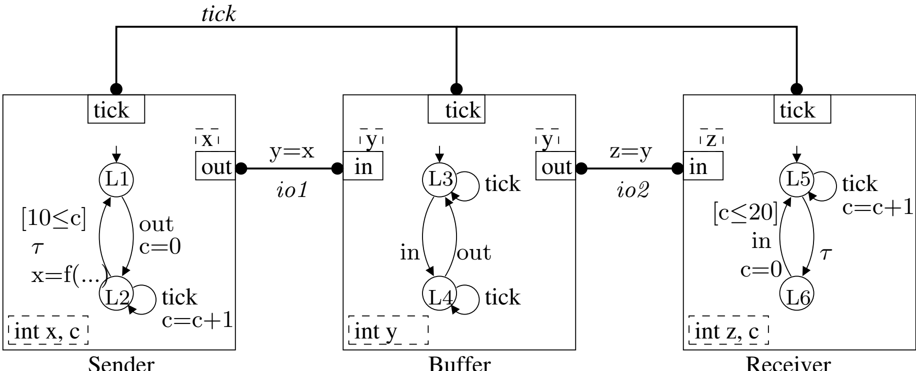


Fig. 1. BIP Example: Sender/Buffer/Receiver System

of *Sender* is described as a transition system with control locations *L1* and *L2*. It communicates through ports *tick* and *out*. Port *out* exports the variable x. Components *Sender*, *Buffer* and *Receiver* are composed by two binary connectors *io1, io2* and a ternary connector *tick*. *tick* represents a rendezvous synchronization between the *tick* ports of the respective components. *io1* represents an interaction with data transfer from the port *out* of *Sender* to the port *in* of *Buffer*. As a result of the data transfer associated with *io1*, the value of variable x of *Sender* is assigned to the variables y of the *Buffer*.

BIP is supported by a rich toolset[1] which includes tools for checking correctness, for source-to-source transformations and for code generation. Correctness can be either formally proven using invariants and abstractions, or tested by using simulation. For the latter case, simulation is driven by a specific middleware, the BIP engine, which allows to explore and inspect traces corresponding to BIP models. Source-to-source transformations allow to realize static optimizations as well as specific transformations towards implementation i.e., distribution. Finally, code generation targets different platforms and operating systems support (e.g., distributed, multi-threaded, real-time, for single/multi-core platforms, etc.).

3 BIP Design Flow for Manycore

The BIP design flow uses a single language to ensure consistency between the different design steps. This is mainly achieved by applying source-to-source transformations between refined system models. These transformations are proven correct-by-construction, that means, they preserve observational equivalence and consequently essential safety properties. The design flow involves several distinct steps, as illustrated in figure 2 and explained below:

1. The *translation* of the application software into a BIP model. This allows its representation in a rigorous semantic framework. Translations for several

programming models (including synchronous, data-flow and event-driven) and domain specific languages into BIP are defined and implemented.
2. *Correctness checking of functional properties* of the application software. Functional verification needs to be done only on high-level models since safety properties and deadlock-freedom are preserved by different transformations applied along the design flow. To avoid inherent complexity limitations, the verification method relies on compositionality and incremental techniques.
3. The *construction of an abstract system model.* This model is automatically generated from 1) the BIP model representing the application software; 2) a BIP model of the target execution platform; 3) a mapping of the atomic components of the application software model into processing elements of the platform. The abstract system model takes into account hardware constraints such as various latencies, mutual exclusion induced from sharing physical resources (like buses, memories and processors) as well as scheduling policies seeking optimal use of these resources.
4. The *construction of a distributed system model.* This model is automatically generated from the abstract system model by expressing high-level coordination mechanisms e.g., interactions and priorities, in terms of primitives of the execution platform. This transformation involves the replacement of atomic multiparty interactions and/or dynamic priorities by protocols using asynchronous message passing (send/receive primitives) and arbiters ensuring semantics preservation. These transformations are proved correct-by-construction [14].
5. The *generation of platform dependent code*, including both functional and glue code for deploying and running the application on the target manycore. In particular, components mapped on the same core can be statically composed thus avoiding extra overhead for (local) coordination at runtime.
6. The *calibration* step, which consists in estimating execution times of actions of the distributed system model. These are obtained through execution and profiling of code fragments compiled on the target platform. They are used to obtain an instrumented system model which takes into account dynamic behavior of the execution platform.
7. The *performance analysis* step involving simulation-based methods combined with statistical model checking on the instrumented system model.

Some of the steps of the design flow are detailed hereafter. We focus on the translation of the application software in BIP (step 1), functional correctness checking by using D-Finder (step 2), platform dependent code generation (step 5), calibration and performance analysis (steps 6 and 7). The construction of the abstract system model (step 3) is presented in [15]. A complete presentation of transformations for building distributed system models (step 4), ready for implementation on distributed platforms, can be found in [14].

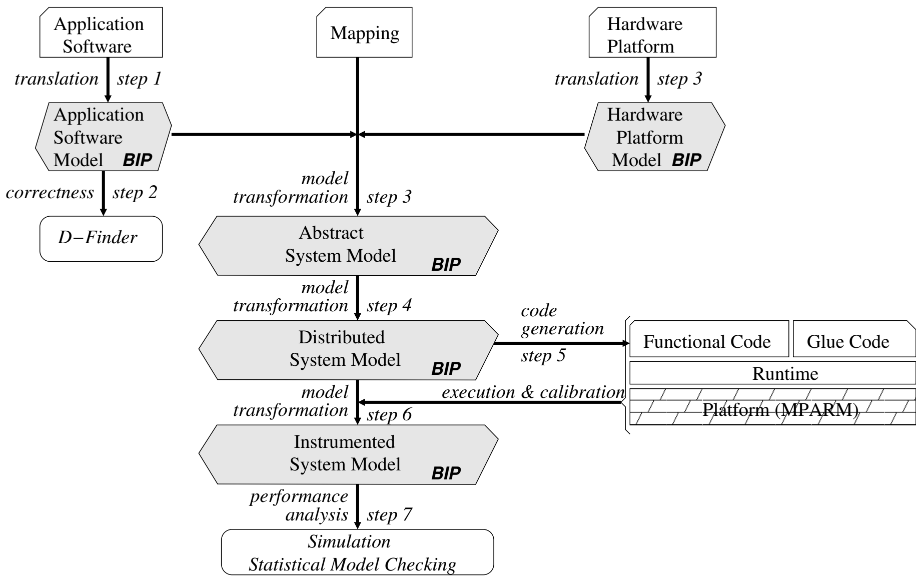


Fig. 2. BIP Design Flow for Manycore

3.1 Translating Application Software into BIP

The first step in the design flow requires the generation of a BIP model for the application software. We have developed a general method for generating BIP models from languages with well-defined operational semantics. We have implemented BIP model generators for several programming models and languages such as Lustre, Simulink and NesC/TinyOS. In this paper, we focus on applications described in the DOL (Distributed Operation Layer) framework [35].

An application software in DOL is a Kahn process network that consists of three basic entities: *Processes*, *FIFO* channels, and *Connections*. The network structure is described in XML. Processes are defined as sequential C programs with a particular structure. For a process P, its state is defined as an arbitrary C data structure named *P_state* and its behavior as the program *P_init(); while (true) P_fire();* where *P_init(), P_fire()* are arbitrary functions operating on the process state. Communication is realized by two primitives, namely *write* and *read* for respectively sending and receiving data to FIFO channels. Moreover, the *P_fire()* method invokes a *detach* primitive in order to terminate the execution of the process.

The construction of the application software model in BIP is done through translation of the above entities in BIP. The construction is structure-preserving: every process and every FIFO are independently translated into atomic components in BIP and then connected according to the connections in the process network [15]. The translation of process behavior requires extraction of an explicit control flow graph from the C code and its representation as an atomic

component in BIP. A FIFO channel is translated into a predefined BIP atomic component.

Example 2. The C description of a DOL process is presented in Figure 3. This process belongs to a process network used for the Cholesky factorization experiment, presented later in section 4. The BIP atomic component generated from the DOL process is shown in figure 4. It has ports *IN_SPLIT*, *IN_2_1*, *OUT_JOIN*, control locations *L1* ... *L6* and variables *index*, *len*, *A*, *L* and *X*. Transitions are labeled by ports *IN_SPLIT*, *IN_2_1*, *OUT_JOIN* and β (internal).

```
void p_2_2_init(DOLProcess *p) {
  p->local->index = 0;
  p->local->len = LENGTH; }
int p_2_2_fire(DOLProcess *p) {
  if (p->local->index < p->local->len) {
    // read input block A22 from splitter
    read((void*)IN_SPLT, p->local->A,
      (K)*(K)*sizeof(double), p);
    // read result block L21 from P21
    read((void*)IN_2_1, p->local->X,
      (K)*(K)*sizeof(double), p);
    // compute A22 = A22 - L21 x L21^t
    SubtractTProduct(p->local->A,
        p->local->X, p->local->X);
    // compute L22 = seq-cholesky(A22)
    Cholesky(p->local->L, p->local->A);
    // send the result L22 to the joiner
    write((void*)OUT_JOIN, p->local->L,
      (K)*(K)*sizeof(double), p);
    p->local->index++; }
  else {
    // termination
    detach(p);
    return -1; }
  return 0; }
```

Fig. 3. DOL Process Description in C

Fig. 4. Translation as BIP Atomic Component

3.2 Checking Application Correctness

The BIP design flow includes a verification step for checking essential functional properties. Application software models in BIP are verified by using the D-Finder tool[10,11]. D-Finder implements compositional methods for generation of invariants and verification of safety properties, including deadlock-freedom.

In general, compositional verification techniques [5,2,17,16,21,27,31,34] are used to cope with state explosion in concurrent systems. The idea is to apply divide-and-conquer approaches to infer global properties of complex systems from properties of their components. Separate verification of components limits state explosion. Nonetheless, designing compositional verification techniques

is difficult since components mutually interact in a system and their behavior and properties are inter-related. As explained in [24], compositional rules are in general of the form:

$$\frac{B_1 < \Phi_1 >,\ B_2 < \Phi_2 >,\ C(\Phi_1, \Phi_2, \Phi)}{B_1 \| B_2 < \Phi >} \tag{1}$$

That is, if two components with behaviors B_1, B_2 meet individually properties Φ_1, Φ_2 respectively, and $C(\Phi_1, \Phi_2, \Phi)$ is some condition taking into account the semantics of parallel composition operation and relating the individual properties with the global property, then the system $B_1 \| B_2$ resulting from the composition of B_1 and B_2 will satisfy a global property Φ.

D-Finder[10,11] provides a novel approach for compositional verification of invariants in BIP based on the following rule:

$$\frac{\{B_i < \Phi_i >\}_{i=1}^n,\ \Psi \in II(\|_\gamma, \{B_i\}_{i=1}^n, \{\Phi_i\}_{i=1}^n),\ (\bigwedge_i \Phi_i) \wedge \Psi \Rightarrow \Phi}{\gamma(B_1, \cdots B_n) < \Phi >} \tag{2}$$

The rule (2) allows to prove a global invariant Φ for a composite component $\gamma(B_1, \ldots, B_n)$, obtained by composing a set of atomic components $B_1, ..., B_n$ by using a set of interactions γ. The premises ensure respectively that, Φ_i is a local invariant of component B_i for every $i = 1, \cdots n$ and Ψ is an interaction invariant of $\gamma(B_1, \ldots, B_n)$ computed automatically from interactions γ, components B_i and local invariants Φ_i. D-Finder provides methods for computing both component invariants and interaction invariants as follows:

- *Invariants for atomic components* are generated by static forward analysis of their behavior. D-Finder uses different strategies which allow to derive local assertions, that is, predicates attached to control locations and which are satisfied whenever the computation reaches the corresponding control location. These assertions are obtained through syntactic analysis of the predicates occuring in guards and actions [12].
- *Interaction invariants* express global synchronization constraints between atomic components. Their computation consists of the following steps. First, for given component invariants Φ_i of the atomic components B_i, we compute a finite-state abstractions $B_i^{\alpha_i}$ of B_i where α_i is the abstraction induced by the elementary predicates occurring in Φ_i. This step is necessary only for components B_i which are infinite state. Second, the composition $\gamma(B_1^{\alpha_1}, \cdots, B_n^{\alpha_n})$ which is an abstraction of $\gamma(B_1, \cdots, B_n)$, can be considered as a 1-safe finite Petri net. The set of structural invariants (traps and locks) and linear invariants of this Petri net defines a global abstract interaction invariant, which is computed symbolically by D-Finder. Finally, the concretization of this invariant gives an interaction invariant of the original system.

D-Finder relies on a semi-algorithm to prove invariance of Φ by iterative application of the rule (2). The semi-algorithm takes a composite component $\gamma(B_1, \ldots, B_n)$ and a predicate Φ. It iteratively computes invariants of the form $\mathcal{X} = \Psi \wedge (\bigwedge_{i=1}^{n} \Phi_i)$ where Ψ is an interaction invariant and Φ_i an invariant of component B_i. If $\mathcal{X}$ is not strong enough for proving that Φ is an invariant ($\mathcal{X} \wedge \neg\Phi = false$) then either a new iteration with stronger Φ_i is started or the algorithm stops. In this case, we cannot conclude about invariance of Φ.

Checking global deadlock-freedom of a component $\gamma(B_1, \ldots, B_n)$ is a particular case of proving invariants - proving invariance of the predicate $\neg DIS$, where DIS is the set of the states of $\gamma(B_1, \ldots, B_n)$ from which all interactions are disabled.

3.3 Platform Dependent Code Generation

The design flow provides the facility for generating code for the MPARM platform [9] from distributed system models in BIP. The generated code is targeted for a runtime called *Native Programming Layer* (NPL) implemented for MPARM. The runtime provides APIs for thread management, memory allocation, communication and synchronization. The code generation consists of two parts, the generation of the functional code and the generation of the glue code.

The functional code is generated from the application components consisting of processes and FIFOs. Processes are implemented as threads, and FIFOs are implemented as shared *queue* objects provided by the NPL library. Each process component is translated into a thread. The implementation in C contains the thread local data, queue handles and the routine implementing the specific thread functionality. The latter is a sequential program consisting of plain C computation statements and communication calls (e.g., *queue* API) provided by the runtime. A *read* transition is substituted by a *pop* API call on the respective queue handle. Similarly a *write* transition is substituted by a *push* API call on its respective queue handle.

The glue code implements the deployment of the application to the platform, i.e., allocation of threads to cores and the allocation of data to memories. The glue code is essentially obtained from the mapping. Threads are created and allocated to cores according to the process mapping. Data allocation deals with allocation of the thread stacks and allocation of FIFO queues for communication. In particular, for MPARM deployment, every thread stack is allocated into the $L1$ memory of the core to which the thread is deployed. Queue handles and queue objects are allocated from the cluster shared $L2$ memory. All these operations are implemented by using the API provided by the runtime.

The code generator has been fully integrated into a tool-chain and connected to the BIP system model generation flow. The generated code is compiled by the `arm-gcc` compiler. The compiled code is linked with the runtime library to produce the binary image for execution on the MPARM virtual simulator.

3.4 System Level Modeling and Performance Analysis

In the BIP design flow, system models are used to integrate the (extra-functional) hardware constraints into the software model according to some chosen deployment mapping. The abstract system model is constructed through a series of transformations from the BIP models of respectively the application software and the hardware platform. These two models are *composed* according to the mapping. The construction has been introduced in [15]. The transformations preserve functional properties of the application software model.

The abstract system model is then transformed for distributed implementation and progressively refined by including timing constraints for execution on the chosen platform. These constraints define execution times for elementary functional blocks, that is, BIP transitions within the application software model. More precisely, execution times are measured by running the executable code on MPARM. We measure the CPU time spent by each process performing blocks of computations. This is done by instrumenting the generated code with profiling API provided by the runtime. The API provides cycle accurate estimates for executing a block of code in each processor.

The instrumented system model is therefore used to analyze non-functional properties such as contention for buses and memory accesses, transfer latencies, contention for processors, etc. In the BIP design flow, these properties are evaluated by simulation of the system model extended with observers. Observers are regular BIP components that sense the state of the system model and collect pertinent information with respect to relevant properties i.e., delay for particular data transfers, blocking time on buses, etc. Actually, we provide a collection of predefined observers monitoring and recording specific information for most common non-functional properties.

Simulation is performed by using the native BIP simulation tool[1]. The BIP system model extended with observers is used to produce simulation code that runs on top of the BIP engine, that is, the middleware for execution/simulation of BIP models. The outcome of the simulation with the BIP engine is twofold. First, the information recorded by observers can be used as such to gain insight about the properties of interest. Second, the same information can be used to build much simpler, abstract stochastic models. These models can be further used to compute probabilistic guarantees on properties by using statistical-model checking. This two-phase approach combining simulation and statistical model-checking has been successfully experimented in a different context[7]. It is fully scalable and allows (at least partially) overcoming the drawbacks related to simulation-based approaches, that is, long simulation times and lack of confidence in the obtained results.

4 Experiments

In this section, we report results about implementation and performance evaluation of two applications using the BIP design flow. We consider Cholesky factorization, a useful inverse-like operation on particular matrices, and MJPEG

decoding, a streaming application for decoding of video streams. For both applications, we target the MPARM platform, which is a highly customizable, experimental, many-core platform available in the PRO3D project.

4.1 MPARM Platform

The MPARM [9] platform is a virtual ARM-based multi-cluster manycore platform. It is configured by the number of clusters, the number of ARM cores per cluster, and the interconnect between the clusters. The MPARM simulator allows experimentation with at most four clusters, each with eight ARM7-TDMI processors. The clusters are connected through a 2×2 NoC interconnect. The architecture is shown in Figure 5. Inside a cluster, each ARM core is connected with its private ($L1$) memory through a local bus. There is also a shared cluster memory ($L2$) which is connected with the cores through a cross-bar interconnect. A NoC-based infrastructure is used for inter-cluster communication, which consists of a router, a link, and the network interface (NI) of the individual clusters. The simulator provides cycle-accurate measurements for the execution on the virtual platform. Henceforth, we will use the term MPARM execution to denote execution on the MPARM virtual simulator.

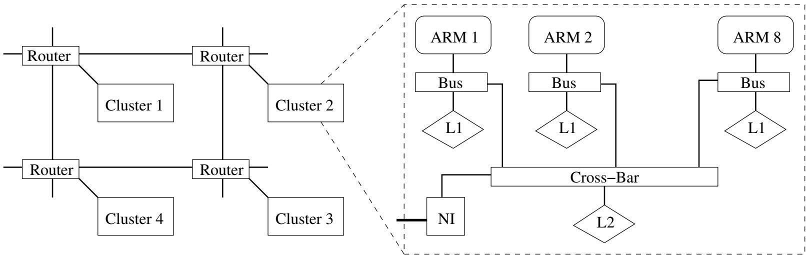


Fig. 5. An MPARM architecture with four clusters

As input to our design flow, we have used the hardware model in BIP generated from a structural description in DOL. The DOL description of the hardware architecture specifies resources connected by communication paths. Resources are of type computation (processors, memories) or communication (buses, crossbar interconnect, routers and links, etc.). Communication paths define the connections between the resources. A part of the DOL description of MPARM is given in Figure 6.

4.2 Cholesky Factorization

Cholesky Factorization decomposes a Hermitian positively-defined real-valued matrix A into the product $L \cdot L^T$ of a lower triangular real-valued matrix L and its conjugate transpose L^T. The Cholesky decomposition is used for solving numerically linear equations $Ax = b$. If A is symmetric and positive definite, then

```
<cluster name="C1" type="MPARM">
    <processor name="P1" type="ARMv7">
       <memory name="Private" type="L1">
            <configuration name="cycles" value="1"/>
       </memory>
       <hw_channel name="local" type="Bus">  </hw_channel>
    </processor>
    . . .
    <processor name="P8" type="ARMv7">
       <memory name="Private" type="L1">
            <configuration name="cycles" value="1"/>
       </memory>
       <hw_channel name="local" type="Bus">  </hw_channel>
    </processor>
    <hw_channel name="X-bar" type="CrossBar">
        <configuration name="cyclesperbyte" value="1"/>
    </hw_channel>
    <memory name="Shared" type="L2">
        <configuration name="cyclesperbyte" value="2"/>
    </memory>
</cluster>
```

Fig. 6. Fragment of the DOL description of an MPARM cluster

we can solve $Ax = b$ by first computing the Cholesky decomposition $A = L \cdot L^T$, then solving $Ly = b$ for y, and finally solving $L^T x = y$ for x.

The sequential Cholesky factorization algorithm has computational complexity $\mathcal{O}(N^3)$ for matrices of size $N \times N$. In this paper, our starting point is the sequential right-looking block-based version [25] provided as algorithm 1 which provides immediate support for parallelization. In this algorithm, B denotes the number of blocks composing the original matrix A, that is $A = (A_{ij})_{1 \leq j \leq i \leq B}$ and every A_{ij} is a block matrix of size $K = N/B$. The algorithm computes the matrix L, block by block, such that $A = L \cdot L^T$. The algorithm 1 is easily parallelizable by separating computations related to different ij-blocks on different processes P_{ij}. Nevertheless, interactions between these processes are highly non-trivial. There are complex patterns for data dependencies, as illustrated in Figure 7 for the cases $B = 2, 3, 4$. Moreover, the amount of computation carried by each process is different. That is, as factorization proceeds, processes with higher indexes (i, j) become computationally more intensive. Furthermore, both data dependencies and the local amount of computation are tightly related to the decomposition size B as well as to the block size K. Altogether, finding optimal implementation on multi-processor platforms with fixed communication and computation resources is a non-trivial problem.

For every B, we denote by *Cholesky(B)* the Cholesky factorization using a $B \times B$ block decomposition. For our experiments, we implemented three versions in DOL, for respectively $B = 2, 3, 4$. In all cases, the process networks contain a *Splitter* process, a *Joiner* process and the computational processes for each block $(P_{ij})_{1 \leq j \leq i \leq B}$. Process *Splitter* splits the initial A matrix into blocks

Algorithm 1. Right-Looking Block-Based Cholesky Factorization

Require: *A Hermitian, positive definite matrix*
Ensure: $A = L \cdot L^T$, *L lower triangular*
  for $k = 1$ **to** B **do**
    $L_{kk} := \mathit{seq\text{-}cholesky}(A_{kk})$
    $L_{kk}^{-T} := \mathit{invert}(\mathit{transpose}(L_{kk}))$
    for $i = k + 1$ **to** B **do**
      $L_{ik} := A_{ik} \cdot L_{kk}^{-T}$
    end for
    for $j = k + 1$ **to** B **do**
      $L_{jk}^T := \mathit{transpose}(L_{jk})$
      for $i = j$ **to** B **do**
        $A_{ij} := A_{ij} - L_{ik} \cdot L_{jk}^T$
      end for
    end for
  end for

and dispatches them to computational processes. Every process P_{ij} implements the computation required on its corresponding matrix blocks A_{ij} and L_{ij}. As an example, the computational processes for *Cholesky(4)* are $P_{11}, P_{21}, P_{22}, P_{31} \ldots P_{44}$ as shown in Figure 7. The final L matrix is re-constructed by the *Joiner* process. Explicit communication between P_{ij} processes is used to enforce data dependencies. In these models, a dedicated FIFO is used for every pair of dependent processes to transfer the result block from the source to the target process. In the MPARM implementation, each computational process is deployed into an ARM processor and all the FIFO buffers are allocated to the $L2$ shared memory. It is to be noted that for $B = 2, 3$ the implementation fits into a single cluster,

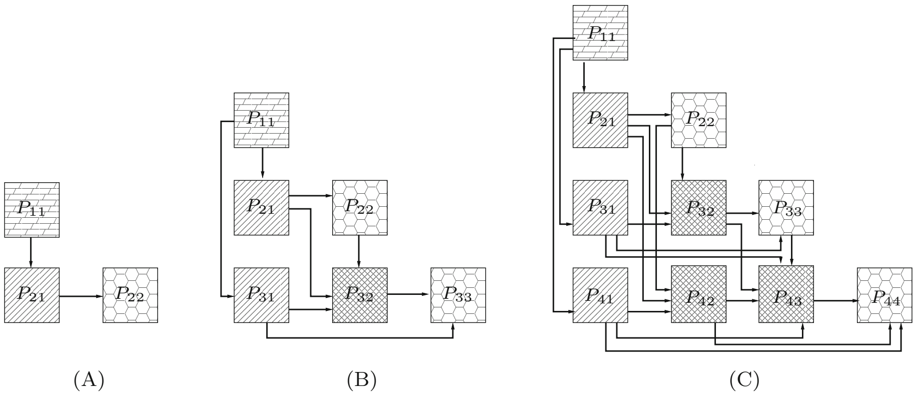


Fig. 7. Data dependencies for $2 \times 2(A), 3 \times 3(B)$ and $4 \times 4(C)$ process decomposition. Identical patterns indicate respectively a similar amount of local computation (processes) or potential for parallel communication (data dependencies).

Table 1. DOL, BIP Models and MPARM Implementation Characteristics

		$B = 2$	$B = 3$	$B = 4$
DOL Process Network	# processes	5	8	12
	# FIFOs	8	20	40
	# lines of code	864	1400	2171
BIP System Model	# components	40	120	181
	# interactions	182	445	882
	# lines of code	5207	7491	13648
MPARM implementation	# lines of code	1977	3163	4923

Table 2. Execution times for computational routines on matrix blocks (in 10^6 cycles)

	$B = 2$	$B = 3$	$B = 4$
	$K = 30$	$K = 20$	$K = 15$
seq-cholesky	33.82	15.47	14.94
invert	34.85	16.06	15.47
transpose	0.13	0.08	0.08
multiply	115.64	53.23	47.16
tmultiply	104.80	45.01	34.89
subtract	1.66	1.05	1.05

and for $B = 4$, two clusters have been used. The magnitude of the different representations produced along the BIP design flow (number of processes, FIFOs, components, interactions, lines of code) is depicted in Table 1.

For every $B = 2, 3, 4$, we evaluate *Cholesky(B)* on 60×60 input matrices of double precision floating point numbers. Therefore, computational processes operate on matrix blocks of size 30×30, 20×20 and 15×15 for respectively $B = 2, 3, 4$. During the calibration phase, each computational routine on matrix blocks is characterized by the number of cycles required to execute it on an ARM processor. This is done by running the generated application code on MPARM and by accurate measurement of the number of cycles, for each routine. Table 2 reports the worst case execution times for different size of matrix blocks.

Table 3 presents an overview of the system-level performance analysis results obtained using two methods, respectively simulation of the system model *vs.* implementation and measurement of code execution on the MPARM platform. For both methods, we report the *total execution time* taken by the application to run on the platform and the *analysis time*, that is, the time taken by the methods to produce the results. We point out that simulation of BIP system models produces fairly accurate results (max 20.95% relative error with respect to the cycle-accurate MPARM execution) while significantly reducing the analysis time (up to 19 times, in some situations). Note that for $B = 4$, the MPARM simulation did not terminate in 72 hours and the simulation data is unavailable. However, an estimate is obtained from the BIP system model simulation. A higher cycle count reflects the communication overhead due to the presence of two clusters with the NoC interconnect.

Table 3. Performance Analysis: MPARM Execution *vs* BIP System Model Simulation

		$B = 2$	$B = 3$	$B = 4$
Total Execution Time (in 10^6 cycles)	MPARM Execution	317.70	229.58	-
	BIP System Model Simulation	325.23	277.69	356.00
	Accuracy	2.37%	20.95%	-
Analysis Time (in minutes)	MPARM Execution	$69'49''$	$34'25''$	-
	BIP System Model Simulation	$3'43''$	$7'54''$	$26'5''$
	Speed-up	18.78	4.35	-

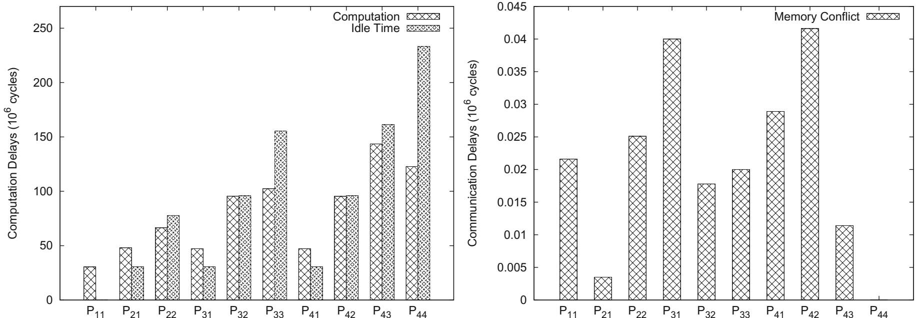


Fig. 8. Performance Results of Computational Processes in *Cholesky(4)*

Finally, Figure 8 presents a detailed view of execution times and communication delays for computational processes for *Cholesky(4)*. For each process, the idle time denotes the waiting time spent before it gets access to read or write on FIFO channels. The communication time denotes the time effectively spent on reading or writing. The computation time denotes the total execution time without the idle and the communication time. The figure 8 (left) confirms that processes with higher indexes (i, j) are indeed computationally more intensive than the others. Additionally, the same processes are also idle for longer time than the others. This happens because of an increased number of data dependencies from processes with lower indexes (i, j). Communication time is impacted by memory conflicts. Memory conflicts occur when two different processes try to access simultaneously FIFO buffers located in the same shared memory. Figure 8 (right) depicts the delays due to memory conflicts for each process.

4.3 MJPEG Decoding

The MJPEG decoder application software reads a sequence of JPEG frames and displays the decompressed video frames. The process network of the application software is shown in Figure 9. It contains five processes *SplitStream* (*SS*), *SplitFrame* (*SF*), *IqzigzagIDCT* (*IDCT*), *MergeFrame* (*MF*) and *MergeStream* (*MS*). The DOL description of the application processes contains approximately 1600 lines of C code.

Fig. 9. Process Network of the MJPEG Decoder Application

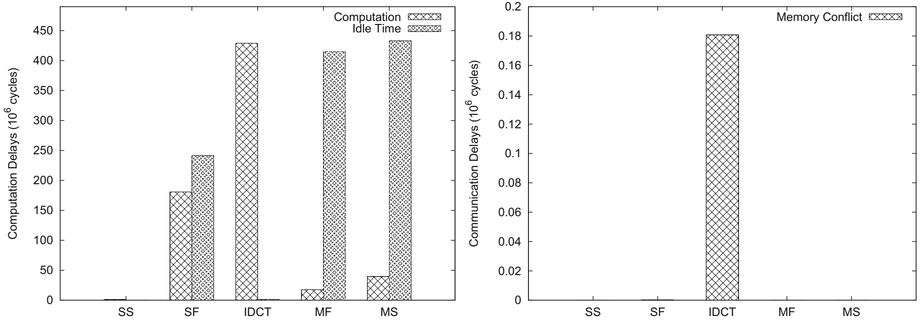


Fig. 10. Performance Results of Computational Processes in MJPEG Decoder

The system model in BIP contains 42 atomic components with 198 interactions, and consists of approximately 7325 lines of BIP code. The implementation generated for MPARM is approximately 3174 lines of code.

For the experiments, we mapped the application on a single MPARM cluster. Each computational process is deployed into an ARM processor and all the FIFO buffers are allocated to the $L2$ shared memory. The performance results per process obtained by simulation of the system model are depicted in Figure 10. We remark that process *IqzigzagIDCT* is the heaviest in terms in computation, while process *MergeStream* stays idle most of the time. The low values of memory conflicts highlights the restricted parallelism within the application.

At system level, we measured the total execution time needed for the decompression of a single frame. Using BIP system model simulation, this time is estimated at 472.88 Mcycles. This result is very close to the cycle-accurate value obtained by measuring the MPARM execution, which is 468.83 Mcycles. The relative error of our estimation is therefore less than 0.87%. Regarding analysis time, BIP system model simulation outperforms execution on (virtual) MPARM. The former completes in $9'46''$ and is approximately 5.2 times faster than the second, which completes in $50'48''$.

The above experiments show the capability of the BIP design flow for fine grain performance analysis on manycore platforms. It also shows the speedup compared to simulation based techniques, without adversely affecting the accuracy of the measurements.

5 Discussions

The presented method allows generation of a correct-by-construction system model for manycore platforms from an application software and a mapping. The

method is based on source-to-source correct-by-construction transformation of BIP models. It is completely automated and supported by the BIP toolset. The system model is obtained by first refining the application software model and then composing it with the hardware architecture model. The composition is defined by the mapping. The construction of the system model is incremental and structure-preserving. This ensures scalability as the complexity of system models increases polynomially with the size of the application software and of the target hardware architecture. Mastering system model complexity is achieved thanks to the expressiveness of the BIP modeling framework.

The method clearly separates software and hardware design issues. It is also parameterized by design choices related to resource management such as scheduling policies, memory size and execution times. This allows estimation of the impact of each parameter on system behavior. Using BIP as a unifying modeling formalism for both hardware and software confers multiple advantages, in particular rigorousness. The obtained system models are correct-by-construction. This is a main difference from other ad hoc model construction techniques.

When the generated system model is adequately instrumented with execution times, it can be used for performance analysis and design space exploration. Experimental results show the feasibility of the approach for fine grain analysis of architecture and mapping constraints on system behavior. The method is tractable and allows design space exploration to determine optimal solutions.

References

1. `http://www-verimag.imag.fr/bip-tools,93.html`
2. Abadi, M., Lamport, L.: Conjoining specifications. ACM Transactions on Programming Languages and Systems 17(3), 507–534 (1995)
3. Abdeddaim, Y., Asarin, E., Maler, O.: Scheduling with Timed Automata. Theoretical Computer Science 354, 272–300 (2006)
4. Henia, R., et al.: System-level performance analysis - the SymTA/S approach. In: IEEE Proceedings Computers and Digital Techniques, vol. 152, pp. 148–166 (2005)
5. Alur, R., Henzinger, T.: Reactive modules. In: Proceedings of LICS 1996, pp. 207–218. IEEE Computer Society Press (1996)
6. Basu, A., Bozga, M., Sifakis, J.: Modeling Heterogeneous Real-time Systems in BIP. In: Proceedings of SEFM 2006, pp. 3–12. IEEE Computer Society Press (2006)
7. Basu, A., Bensalem, S., Bozga, M., Caillaud, B., Delahaye, B., Legay, A.: Statistical Abstraction and Model-Checking of Large Heterogeneous Systems. In: Hatcliff, J., Zucca, E. (eds.) FMOODS/FORTE 2010, Part II. LNCS, vol. 6117, pp. 32–46. Springer, Heidelberg (2010)
8. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous component-based design using the BIP framework. IEEE Software, Special Edition – Software Components beyond Programming – from Routines to Services 28(3), 41–48 (2011)
9. Benini, L., Bertozzi, D., Bogliolo, A., Menichelli, F., Olivieri, M.: MPARM: Exploring the Multi-Processor SoC Design Space with SystemC. Journal of VLSI Signal Processing Systems 41, 169–182 (2005)

10. Bensalem, S., Bozga, M., Sifakis, J., Nguyen, T.-H.: Compositional Verification for Component-Based Systems and Application. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 64–79. Springer, Heidelberg (2008)
11. Bensalem, S., Bozga, M., Nguyen, T.-H., Sifakis, J.: D-Finder: A Tool for Compositional Deadlock Detection and Verification. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 614–619. Springer, Heidelberg (2009)
12. Bensalem, S., Lakhnech, Y.: Automatic generation of invariants. FMSD 15(1), 75–92 (1999)
13. Bensalem, S., Bozga, M., Legay, A., Nguyen, T.H., Sifakis, J., Yan, R.: Incremental Component-based Construction and Verification using Invariants. In: Proceedings of FMCAD 2010, pp. 257–266. IEEE (2010)
14. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: A Framework for Automated Distributed Implementation of Component-based Models. Distributed Computing (to appear, 2012)
15. Bourgos, P., Basu, A., Bozga, M., Bensalem, S., Sifakis, J., Huang, K.: Rigorous system level modeling and analysis of mixed HW/SW systems. In: Proceedings of MEMOCODE 2011, pp. 11–20. IEEE/ACM (2011)
16. Chandy, K., Misra, J.: Parallel program design: a foundation. Addison-Wesley Publishing Company (1988)
17. Clarke, E., Long, D., McMillan, K.: Compositional model checking. In: Proceedings of LICS 1989, pp. 353–362 (1989)
18. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity: The Ptolemy approach. Proceedings of the IEEE 91(1), 127–144 (2003)
19. Erbas, C., Pimentel, A.D., Thompson, M., Polstra, S.: A framework for system-level modeling and simulation of embedded systems architectures. EURASIP Journal on Embedded Systems 2007 (2007)
20. Feiler, P.H., Lewis, B., Vestal, S.: The SAE Architecture Analysis and Design Language (AADL) Standard: A basis for model-based architecture-driven embedded systems engineering. In: Proceedings of RTAS Workshop on Model-driven Embedded Systems, pp. 1–10 (2003)
21. Grumberg, O., Long, D.E.: Model checking and modular verification. ACM Transactions on Programming Languages and Systems 16(3), 843–871 (1994)
22. Kienhuis, B., Deprettere, E., Vissers, K., van der Wolf, P.: An approach for quantitative analysis of application-specific dataflow architectures. In: Proceedings of ASAP 1997, pp. 338–349. IEEE Computer Society (1997)
23. Künzli, S., Poletti, F., Benini, L., Thiele, L.: Combining Simulation and Formal Methods for System-level Performance Analysis. In: Proceedings of DATE 2006, pp. 236–241 (2006)
24. Kupferman, O., Vardi, M.Y.: Modular Model Checking. In: de Roever, W.-P., Langmaack, H., Pnueli, A. (eds.) COMPOS 1997. LNCS, vol. 1536, pp. 381–401. Springer, Heidelberg (1998)
25. Leary, D.P., Stewart, G.: Data-flow algorithms for parallel matrix computations. Communications of the ACM 28(8), 840–853 (1985)
26. Lieverse, P., Stefanov, T., van der Wolf, P., Deprettere, E.: System level design with SPADE: an M-JPEG case study. In: ICCAD, pp. 31–38 (2001)
27. McMillan, K.L.: A Compositional Rule for Hardware Design Refinement. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 24–35. Springer, Heidelberg (1997)

28. Moussa, I., Grellier, T., Nguyen, G.: Exploring SW Performance Using SoC Transaction-Level Modeling. In: Proceedings of DATE 2003, pp. 20120–20125 (2003)
29. Nikolov, H., Thompson, M., Stefanov, T., Pimentel, A., Polstra, S., Bose, R., Zissulescu, C., Deprettere, E.: Daedalus: toward composable multimedia mp-soc design. In: Proceedings of DAC 2008, pp. 574–579. ACM (2008)
30. OMG: OMG Systems Modeling Language SysML (OMG SysML). Object Management Group (2008)
31. Pnueli, A.: In transition from global to modular temporal reasoning about programs, pp. 123–144 (1985)
32. PRO3D: Programming for Future 3D Architecture with Many Cores, FP7 project funded by the EU under grant agreement 248 776, http://pro3d.eu/
33. Salah, R.B., Bozga, M., Maler, O.: Compositional Timing Analysis. In: Proceedings of EMSOFT 2009, pp. 39–48 (2009)
34. Stark, E.W.: A Proof Technique for Rely/Guarantee Properties. In: Maheshwari, S.N. (ed.) FSTTCS 1985. LNCS, vol. 206, pp. 369–391. Springer, Heidelberg (1985)
35. Thiele, L., Bacivarov, I., Haid, W., Huang, K.: Mapping Applications to Tiled Multiprocessor Embedded Systems. In: Proceedings of ACSD 2007, pp. 29–40. IEEE Computer Society (2007)
36. Thiele, L., Chakraborty, S., Naedele, M.: Real-time calculus for scheduling hard real-time systems. In: Proceedings of ISCAS 2002, vol. 4, pp. 101–104. IEEE (2002)

Low-Cost Dynamic Voltage and Frequency Management Based upon Robust Control Techniques under Thermal Constraints

Sylvain Durand[1], Suzanne Lesecq[2], Edith Beigné[2], Christian Fabre[2], Lionel Vincent[2], and Diego Puschini[2]

[1] NECS Team, INRIA/GIPSA-lab joint team
Inovallée, 655 avenue de l'Europe, 38334 Saint Ismier Cedex, France
sylvain@durandchamontin.fr
[2] CEA, LETI MINATEC Campus
17 rue des Martyrs, 38054 Grenoble Cedex 9, France
firstname.lastname@cea.fr

Abstract. Mobile computing platforms need ever increasing performance, which implies an increase in the clock frequency applied to the processing elements (PE). As a consequence, the distribution of a single global clock over the whole circuit is tremendously difficult. Globally Asynchronous Locally Synchronous (GALS) designs alleviate the problem of clock distribution by having multiple clocks, each one being distributed on a small area of the chip. Energy consumption is the main limiting factor for mobile platforms as they are powered by batteries. Dynamic Voltage and Frequency Scaling (DVFS) in each Voltage and Frequency Island (VFI) has proven to be highly effective to reduce the power consumption of the chip while meeting the performance requirements. Environmental parameters (i.e. temperature and supply voltage) changes also strongly affect the chip performance and its power consumption. Some sensors can be buried in order to estimate via data fusion techniques the supply voltage and the temperature variations. For instance the knowledge of the gap between the temperature and its maximum value can be used to adapt the power management technique. The present paper deals with the design of a voltage and frequency management approach (DVFS) that explicitly takes into account the thermal constraints of the platform.

1 Introduction

Mobile computing platforms need ever increasing performance which implies an increase in the clock frequency applied to the processing elements (PE). Unfortunately, this performance increase together with the technology shrinking render the distribution of a single global clock over a digital circuit extremely difficult. The problem of clock distribution can be soften by the design of Globally Asynchronous Locally Synchronous (GALS) architectures: a clock is applied on a small area of the chip (the synchronous domain), and several asynchronous clocks are used to clock the whole chip.

B. Beckert et al. (Eds.): FMCO 2011, LNCS 7542, pp. 334–353, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Energy availability is one of the main limiting factors for mobile platforms as they are powered by batteries. Power management techniques try to provide just enough power to the PE in order to finish the task on its deadline. Actually, dynamic power consumption depends on the clock frequency F but also on the supply voltage V_{dd}. As a consequence, the power consumption can be minimized with an appropriate setting of F and V_{dd}. The power consumption also depends on the threshold voltage V_{th} that can be as well adjusted. Last but not least, it depends on the temperature of the die, as the leakage highly depends on the temperature.

Another limiting factor for mobile platforms is the thermal aspect. Indeed, the circuit temperature has a significant impact on the power consumption and performance, but also on the cooling and packaging costs [1]. Dynamic Thermal Management (DTM) techniques have been developed in order to ensure for the circuit not to reach a prohibitive temperature. The thermal management can be obtained thanks to DVFS approaches but also with task migration among the cores for multicore platforms [2].

In these situations, control algorithms must be developed to ensure that the performance requirements are met while the power consumption is minimized and the temperature of the whole circuit stays below a pre-defined threshold. Moreover, the limitation of spatial and temporal temperature gradients or peaks is of great interest in order to improve the circuit reliability. These algorithms must be simple enough in order to limit the overhead they may introduce. Moreover, even if most of the existing methods address power consumption management and temperature control separately, both the power consumption and the temperature should be jointly controlled in order to meet the goal defined above.

The objective of the present paper is to describe a new way of applying Dynamic Voltage and Frequency Scaling in order to minimize temperature cycling and/or peaks. The voltage and frequency values applied to the so-called Voltage and Frequency Island (VFI) depend on the computational workload. Moreover, instead of applying the different voltage and frequency values in order to meet the task deadline one after the other one, they are applied in a chopped way. Note that this chopping has a beneficial side effect as it smoothes and limits the temperature increase. As a consequence, simple control laws as developed in [3] can be applied at run-time while the temperature increase or decrease is limited when a new task is run.

The rest of the paper is organized as follows. Section 2 gives a short overview of recent works in the area of power- and thermal-aware DVFS techniques. Then section 3 provides the problem statement. The power consumption management method based on DVFS approach is described in section 4. For comparison purpose, a "classical" approach is given in section 4.1 while its chopped version that limits temperature increase/decrease is provided in section 4.2. Some simulation results are also presented. Concluding remarks and on-going work highlight are provided in section 5.

2 Survey of Existing Works

Various power management approaches have been described in the literature and a short survey can be found in [4]. Most of these approaches rely on two "actuators", namely, the supply voltage V_{dd} and the clock frequency F whose values are tuned to minimize the power consumption. Note that both values cannot be fixed independently in order to avoid timing faults. These F and V_{dd} control approaches are usually called "Dynamic Voltage and Frequency Scaling" (DVFS). For instance, [3] supposes that several pairs of (F, V_{dd}) values are available for each PE. A predictive control law minimizes the time spent in the most energy-consuming mode under the constraint for the task to be finished on the deadline. The estimation of the switching time is performed at each sampling time, the initial knowledge of the number of instructions to be run being possibly uncertain.

Other works also consider the use of another actuator, i.e. the threshold voltage V_{th}. In [5], three frequency values and a dynamic adaptive biasing are used to manage the power consumption. The values of F and V_{th} are dynamically chosen without any optimization at run-time. The work from Firouzi *et al.* [6] tries to perform a tradeoff between reliability, performance and power consumption using both Body Biasing and DVFS techniques to meet the optimization objective while [7] implements Dynamic Voltage and Threshold Scaling (DVTS) techniques in order to find the optimum power point.

All DVFS and/or DVTS approaches have proven to be highly effective to reduce the power consumption of a digital circuit while meeting the performance requirements [8]. The voltage(s) and frequency are adapted in each VFI [9] of the GALS architecture to minimize the power consumption under performance constraints [10].

Unfortunately, thermal constraints highly limit the use of mobile platforms as the temperature influences both the power consumption and the performance. Moreover, cooling and packaging costs [1] are increased when the temperature of use is higher. Several Dynamic Thermal Management (DTM) techniques have been proposed in the literature, see for instance [11] or [2] and references therein. This latter proposes to control the voltages and frequencies applied to the cores of an heterogeneous multicore platform to guarantee execution of "thermally constrained tasks" with hard deadlines where a thermally constrained task is a task which if executed on a core at the maximum possible speed, the temperature of this core exceeding a given safe temperature limit. The problem is formulated as an optimization one whose objective is to determine the voltages and frequencies applied to the cores such that all tasks running on the cores meet their deadline while satisfying the thermal constraints. The proposed method makes use of accurate power and thermal models, including leakage dependence on temperature, which appears in the set of constraints of the optimization problem. The voltages and speeds are computed so as for the hottest core not to exceed the maximum acceptable temperature. Even if the approach proposed is very appealing, its computational time (in min) is incompatible with an implementation at run-time. Actually, accurate power and thermal models for the underlying

platform must be available and an optimization routine with constraints has to be run at each sampling time.

In [12], the problem of power control under temperature constraints for a multi-processor platform is addressed. A chip-level temperature-constrained power control algorithm based on Model Predictive Control (MPC) theory for Multi-Inputs Multi-Outputs (MIMO) systems is considered. The solution intends to optimize the processor performance while the total power consumption of the circuit is controlled to reach a given setpoint (i.e. the desired constraint) and the temperature is lower than a pre-fixed threshold. Measurements provided by a temperature sensor in each core and a power monitor for the whole circuit are used by the feedback control loop. Moreover, the CPU utilization and the number of instructions for each core are also provided to the centralized controller. The DVFS is implemented with "DVFS levels" that indeed correspond to frequency levels. Moreover, the desired DVFS levels (i.e. the various frequency clocks to be applied to the cores) computed by the MPC are approximated with a sequence of supported DVFS levels (i.e. frequency clocks that are available on the circuit). Note that supply voltage changes do not seem considered in [12] and their power model consumption is supposed linear in the DVFS level. Recall that a Model Predictive Controller optimizes a cost function. The system model is used to predict the control behaviour over a prediction horizon. The control objective is to select an input trajectory that minimizes a cost function while satisfying the constraints. Thus, an optimization problem has to be solved at each sampling time, and the system models must be available.

3 Context and Problem Statement

Due to the complexity and size of today computing platforms, power consumption and/or thermal managements cannot be any more based on open-loop strategies [12]. Moreover, for advanced technologies, the power consumption control must be coupled with thermal aspects as the power leakage highly depends on the temperature. Lastly, the implementation of closed-loop control helps the platform to adapt its functioning to Process, Voltage and Temperature (PVT) variability.

Closed-loop control approaches require the knowledge of the state of the circuit, for instance its internal temperature or the value of the supply voltage V_{dd} really applied to the Voltage Frequency Island, and of the progress of the software that is running on the PE (e.g. the number of instructions already treated by the task that is currently running on the PE). Therefore, sensors must be buried within the chip as well as monitors have to be implemented for the software aspects. Various "environmental" sensors have been developed to monitor V_{dd} and the Temperature T within integrated circuits. They are divided in two sets. The first one corresponds to "specialized" sensors that are designed to be sensitive to only one of the environmental parameters. For instance, they are sensitive to V_{dd} (resp. T) changes [13][14]. Unfortunately, their design is tricky as they must be insensitive to all but one parameter. The second set contains general purpose

sensors, built of standard digital blocks, that are sensitive to several parameters. For example, the oscillating frequency F at a Ring Oscillator (RO) output depends on its current Process-Voltage-Temperature state. A counter associated to this RO provides a raw digital measurement, sensitive to all these parameters [15][16]. The output of this RO cannot directly provide information on one of these parameters. Thus, a set of ROs (e.g. similar to the MultiProbe described in [17]) must be used in conjunction with a fusion technique to monitor the state of the chip [18].

It is well admitted that for CMOS technologies, the dynamic power consumption P is a quadratic function of the supply voltage V_{dd} and a linear function of the clock frequency F [19]:

$$P = KV_{dd}^2F \tag{1}$$

where K mainly depends on the technology. As a consequence, an appropriate choice of V_{dd} and F to perform a given task T_i before its deadline via DVFS techniques [20] clearly optimizes the power consumption. The deadline can be interpreted as a constraint when the problem is solved via optimization techniques. Recall that V_{dd} and F cannot be freely fixed in order to avoid timing faults. The control of the energy-performance tradeoff in a voltage scalable device can hence be formulated as follows (**Problem 1**):

- minimize the energy consumption by the reduction (as much as possible) of the penalizing supply voltage;
- while ensuring a "good" computational performance so that the tasks meet their deadline [21].

Note that when the voltage actuator is realised with a Vdd-hopping technique, the voltage transitions have to be minimized as the actuator will consume more during these transitions, see [22] for further details. A solution for such a problem was for instance proposed in [23] where a control law computes dynamically an energy-efficient speed setpoint (given as the number of instructions per second) that the system has to track in order to satisfy the control objective, by adapting the control variables (i.e. the voltage and frequency levels, afterwards denoted V_{level} and f_{level} respectively). [23] proposes three overlapped control loops applied at different architecture levels[1] to dynamically manage the energy and activity into a digital circuit:

- the inner loop controls the supply voltage V_{dd} and the clock frequency F actuators applied to each PE;
- the second loop, at a higher level, is used to control the tradeoff between the energy consumption and the computational performance in each VFI;
- the external loop deals with the Quality of Service (QoS) at the applicative level. This latter loop runs at the Operating System (OS) or scheduler level. It manages the different VFIs of the chip with their own performance.

[1] This suggestion has been done in the context of the French Minalogic project ARAVIS. Its main goal was to provide some architectural advices and solutions for the power management of computational platforms in advanced technology.

The whole architecture is depicted in Fig. 1. Note that the dynamics of the different loops depends on the integration level. Indeed, the deepest loops exhibit the fastest updates of the control signals, with a gain of 100 to 1000. As a result, the dynamics of the deepest level can be neglected. Consequently, the different levels interact without conflict.

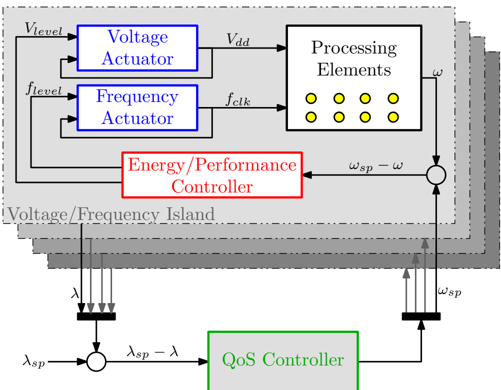


Fig. 1. Power control within a digital circuit, organized with 3 feedback loops

Unfortunately, the approach by Durand *et al.* does not take into account the thermal constrains, and an external mechanism should be added in order to ensure that the temperature T does not exceed its maximum allowed value. Therefore, the power minimization problem must be recast as a new optimization one so as to take into account the thermal constraints (**Problem 2**):

- minimize the energy consumption by the reduction (as much as possible) of the penalizing supply voltage V_{dd};
- while ensuring "good" computational performance so that the tasks meet their deadline [21];
- under thermal constraints.

The objective of this paper is to propose a solution to this problem.

4 Power Consumption Management

The present paper focuses on the energy-performance tradeoff loop of each VFI, see Figure 1. The control signals directly act on the voltage and frequency closed-loop actuators that modify the supply power of the VFI at given transition times. Note that the voltage and frequency actuators are supposed to provide a set of discrete values. The "setpoint" λ is provided by the OS. It is related to the real-time requirements used to define a computational load profile with respect to time.

Here, **Problem 2** is solved via a particular implementation of the DVFS developed for **Problem 1**. Actually, **Problem 1** is easier to solve than **Problem 2** as the thermal constraint depends on the power consumption [2]. Thus a simpler control law is developed for **Problem 1** and the supply voltage values are applied in a chopped way in order to limit the thermal influences. Note that in the solution proposed, an external mechanism must be added in order to be sure that the platform will not exceed its maximum allowed temperature.

Section 4.1 summarizes the results from [3][23] while Section 4.2 extends the previous results in order to limit the thermal effects.

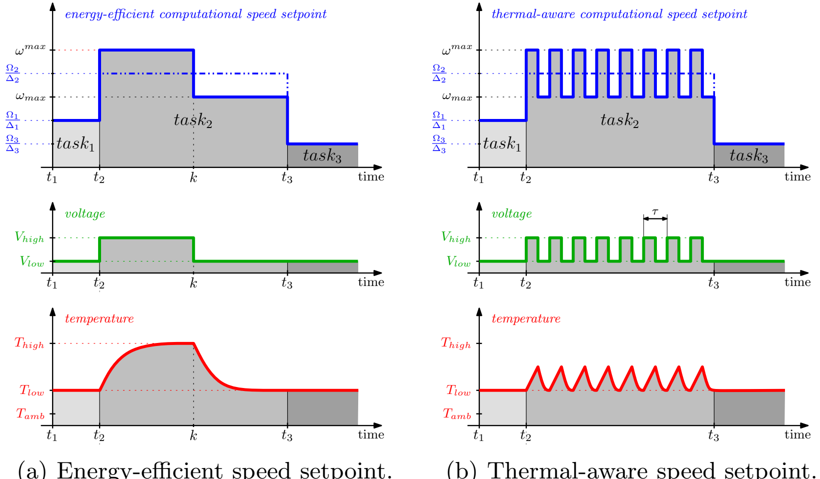
4.1 Control of the Energy-Performance Tradeoff

Durand [3] proposes to control the energy-performance tradeoff of a VFI by the adaptation of the control variables V_{level} and f_{level}. The setpoint is based on some information provided by the OS or scheduler for each task T_i to be run, namely, the number of instructions Ω_i and the remaining available time to complete the task, denoted the laxity Λ_i. In the present paper, two supply voltage values are considered, V_{high} and V_{low}. Let ω^{max} and ω_{max} denote the maximal computational speeds when the system is running at V_{high} and V_{low} respectively. It follows that V_{high} is applied as soon as the average speed setpoint for a task is higher than ω_{max} in order to meet the associated deadline. An intuitive method consists in building the average speed setpoint of each task:

$$\bar{\omega} = \Omega_i / \Delta_i \tag{2}$$

where Δ_i is the deadline associated to task T_i, in such a way that the number of instructions to run is performed at the end of the task. However, this method is not energy-efficient since a whole task can be computed with the penalizing high supply voltage V_{high}. The solution proposed in [3] consists in splitting the tasks into two parts, see Figure 2(a) for task $task_2$. The PE starts to run at V_{high} – if required – with the maximal available frequency in order to achieve the maximal possible speed ω^{max}. Therefore, it runs faster than the average speed $\bar{\omega}$, from time t_2 to k. Then, $task_2$ can be finished at V_{low} with a speed lower than ω_{max}. Only one frequency F_{high} is supposed available when running at V_{high} whereas several frequency levels can be applied to the PE when V_{low} is used. Here, two frequencies F_{low_1} and F_{low_2}, $F_{low_1} > F_{low_2}$, are supposed available. These frequency levels allow the task to finish exactly on its deadline.

The time when the voltage switches from V_{high} to V_{low} has to be suitably calculated by the controller in order to ensure the required computational performance, e.g. the deadline is never missed. However, k is not *a priori* known (as the number of instructions to be run might be not exactly known). Therefore, a predictive control law is used to dynamically calculate the switching time k. Recall that *predictive control* consists in finding a certain control profile over a time horizon to achieve a given objective. The predictive issue can be formulated as an optimization problem. However, this optimal criterion is associated to a high computational cost, which is not acceptable in embedded systems with limited resources.



(a) Energy-efficient speed setpoint. (b) Thermal-aware speed setpoint.

Fig. 2. Different computational speed setpoint profiles: whereas the energy-efficient profile is used to save energy while ensuring good performance, the thermal-aware one allows to reduce the temperature increase as well.

Nevertheless, the strategy adopted here is called *fast predictive control*. It consists in taking advantage of the structure of the dynamical system to fasten the determination of the control profile [24]. Indeed, the closed-loop solution yields an easier and faster algorithm as one simply needs to calculate the speed required to fit the task into its deadline regarding what has already been executed:

$$\delta = \frac{\Omega_i - \Omega}{\Lambda_i} \tag{3}$$

where δ is the *predicted speed* and Ω is the number of instructions executed from the beginning of the task T_i. The energy-efficient speed setpoint is then directly deduced from the value of the predicted speed and so are the voltage and frequency levels.

Indeed, the system has to run at (V_{high}, F_{high}) when the required speed is higher than the maximal speed at low voltage, i.e. $\delta(t_{k+1}) > \omega_{max}$. Otherwise, V_{low} and one of its associated frequency will be applied to finish the task on the deadline. The determination of the frequency level at V_{low} is similar. A so-called *clock-gating* phase [25] can also be applied. In this situation, V_{low} is applied to the processor with a null clock frequency F. This situation is useful to save some power when a task is ended before its deadline. Furthermore, in order to be robust to uncertainties in ω^{max} and ω_{max}, these speeds are estimated using a weighted average of the measured speed. The reader can refer to [23] for a complete presentation of this approach. Few tricks were also suggested in this latter reference to provide a low-cost implementation of the control strategy.

Simulation Results. A scenario with three tasks to be executed is now presented. The number of instruction and deadline for each task are supposed known. The simulation results are depicted in Figure 3. The first plots (top) represent the number of instruction and deadline for each task. The laxity is plotted with the deadline. The plots in the middle of the figure show the average speed setpoint for each task, the predicted speed and the measured computational speed. The bottom plot shows the supply voltage. For this scenario, the system runs during about 80 % of the simulation time at low voltage and a reduction of about 30 % and 65 % of the energy consumption is achieved (in comparison with a system without DVS and DVFS mechanism respectively).

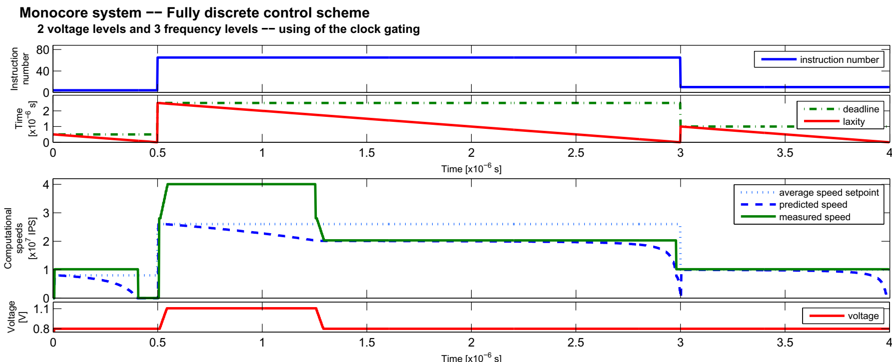


Fig. 3. Simulation results of the energy-performance tradeoff control loop in Figure 1

.

4.2 Thermal-Aware Control of the Energy-Performance Tradeoff

Whereas the previous control strategy manages the tradeoff between energy and performance, it does not consider the thermal effects. The main contribution of the present paper is to overcome this drawback. Note that the thermal dynamics of a circuit can be approximated by a first-order model [26]:

$$\frac{dT(t)}{dt} = aP(t) - b\big[T(t) - T_{amb}\big] \tag{4}$$

where P and T are the power consumption and the temperature of the chip respectively and T_{amb} is the ambient temperature. This equation is similar to a RC electrical circuit where $a = {}^1/_C$ and $b = {}^1/_{RC}$ are some constants related to the thermal system, see Figure 4.

When a power mode (defined with the supply voltage and its associated clock frequency) is applied to the VFI for a long period of time, the temperature within the chip reaches a steady-state value. For the two power modes previously defined, these temperatures are denoted T_{low} and T_{high}. Applying V_{high}

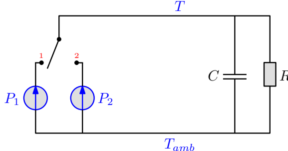


Fig. 4. RC equivalent electrical model of the thermal problem

(and its associated frequency) for a long time period can lead to the occurrence of a (possible) hot spot within the chip, as can be seen in Figure 2(a). Whereas T_{low} cannot be reduced, the temperature achieved at the penalizing high voltage mode can be decreased. The idea previously suggested for instance in [27][26][28][29][30][31] consists in switching between both voltage levels in order not to reach the maximum temperature T_{high}. Note that these papers are restrained to periodic tasks whereas the strategy proposed here can be applied to periodic but also non-periodic tasks. The execution of a task is divided into different periods, as depicted in Figure 2(b). The side effect of this chopping strategy is that the energy consumption will slightly increase because several new transitions are introduced. The control problem hence naturally becomes a tradeoff between energy, performance and temperature.

Furthermore, the power also varies with the temperature due to the leakage term [32][28]. It is hence mandatory to propose a thermal-aware control of the energy-performance tradeoff. Note that in the present work, the leakage power is considered as constant w.r.t. the temperature. The reader is invited to refer to [2] where the dependence of the temperature constraint with respect to the power consumption that itself depends on the temperature is taken into account in the optimization problem. Even if the power modes are applied in a chopped way, the ratio between high and low voltage levels has to remain (for a given task) equal to the one applied with the energy-efficient scheme of Section 4.1 in order to keep the performance. This ratio, denoted hereafter *duty ratio* κ, is computed thanks to the previous predictive control law. Moreover, κ is dynamically updated at the beginning of each *period of oscillation* τ in order to be robust to uncertainties in the task information. First, τ is supposed constant. However, the duty ratio κ can be represented as the distance between the speed to apply until the end of the task – that is the predicted speed δ previously defined in (3) – and the possible maximal speeds at high and low voltage levels:

$$\kappa = \frac{\delta - \omega^{max}}{\omega_{max} - \omega^{max}} \tag{5}$$

The task is hence successively executed at V_{high} and V_{low} during $\kappa\tau$ and $(1-\kappa)\tau$ respectively, and this process is repeated until the end of the task. The average temperature as well as its ripple can be analytically obtained from the RC electrical equivalent model [33] of the thermal equation (4). This is represented in Fig. 4 where the components are assumed to be ideal and, therefore, do not dissipate power.

Suppose that the transition time of the actuators is negligible. Then the different power values P_i are related to different couples (V_i, F_i). Here, P_1 corresponds to (V_{high}, F_{high}) and P_2 is consumed when (V_{low}, F_{low_1}) is applied to the VFI. Moreover, $P_1 \geq P_2$. Note that the other settling points at V_{low} (i.e. corresponding to the different frequency levels that can be applied at V_{low}) are not detailed here but the principle can be easily extended. The temperature increases when P switches from P_2 to P_1 and it decreases otherwise. The average power value depends on the duty cycle of the switching control. This scheme is classical in electronic engineering [34].

Analysis of the Temperature Steady-State Mean Value and Ripple. The thermal equation (4) solution depends on the switch position in Figure 4. The system is supplied by a high power source P_1 (respectively a low power source P_2) when the switch is in position 1 (respectively 2). This behaviour cyclically repeats with a constant period τ, while the duty ratio of this switching control is κ, previously defined in (5). As a consequence, equation (4) becomes:

$$\frac{dT(t)}{dt} = \begin{cases} aP_1 - bT(t) + c & \text{if} \quad 0 \leq t \leq \kappa\tau \\ aP_1 - bT(t) + c & \text{if} \quad \kappa\tau \leq t \leq \tau \end{cases}$$

where $c = bT_{amb}$. The temperature can then be analytically calculated by solving these thermal differential equations, which yields to:

$$T(t) = \begin{cases} k_1 e^{-bt} + T_1 & \text{if} \quad 0 \leq t \leq \kappa\tau \\ k_2 e^{-b(t-\kappa\tau)} + T_2 & \text{if} \quad \kappa\tau \leq t \leq \tau \end{cases} \tag{6}$$

$$\text{with} \quad T_1 := \frac{aP_1 + c}{b} \quad \text{and} \quad T_2 := \frac{aP_2 + c}{b} \tag{7}$$

where $T_1 \geq T_2$ by construction. Furthermore, k_1 and k_2 are some parameters which can also be analytically obtained from:

– the continuous behaviour at $t = \kappa\tau$ for both expressions, and
– the periodicity of the temperature in the steady state, i.e. $T(0) = T(\tau)$.

This finally leads to:

$$\begin{aligned} k_1 &= -\Delta_T \frac{1 - e^{-b(1-\kappa)\tau}}{1 - e^{-b\tau}} \simeq -\Delta_T \left[1 - \kappa + b\frac{\kappa(1-\kappa)}{2}\tau \right] \\ k_2 &= \Delta_T \frac{1 - e^{-b\kappa\tau}}{1 - e^{-b\tau}} \simeq \Delta_T \left[\kappa + b\frac{\kappa(1-\kappa)}{2}\tau \right] \\ \text{with} \quad \Delta_T &:= T_1 - T_2 = \frac{a}{b}(P_1 - P_2) \end{aligned} \tag{8}$$

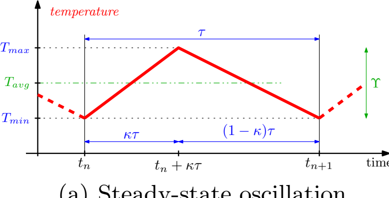
The expressions can be linearised – applying a first order Taylor expansion – since the oscillating period is very small compared with the time constant of the system, i.e. $\tau \ll RC$. Then, the linearised expression becomes

$$T(t) \simeq \mu t + \nu \tag{9}$$

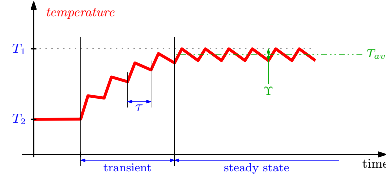
$$\text{with} \quad \mu = \begin{cases} -bk_1 & \text{if} \quad 0 \leq t \leq \kappa\tau \\ -bk_2 & \text{if} \quad \kappa\tau \leq t \leq \tau \end{cases}$$

$$\text{and} \quad \nu = \begin{cases} k_1 + T_1 & \text{if} \quad 0 \leq t \leq \kappa\tau \\ \big(1 + b\kappa\tau\big)k_2 + T_2 & \text{if} \quad \kappa\tau \leq t \leq \tau \end{cases}$$

The waveform of the temperature in steady-state condition is given in Figure 5(a).



(a) Steady-state oscillation.



(b) Transient phase when the power is increased.

Fig. 5. Thermal-aware control behaviour: the system alternatively runs with the switch in position 1 during $\kappa\tau$ and then in position 2 during $(1 - \kappa)\tau$. A certain dynamics is required before the oscillations reach a given steady state where the temperature finally varies with a constant average value T_{avg} and a given ripple Υ.

The temperature starts with the initial value $T(0)$. Then it increases during the first subinterval, when the switch is in position 1, with a positive slope given by μ in (9). At time $t = \kappa\tau$, the switch changes from position 1 to position 2, and the temperature decreases with a negative slope. At time $t = \tau$, the switch changes back to position 1 and the process repeats. The average value of the temperature as well as its ripple can be easily computed. Both depend on the minimum and maximum temperature peak values, denoted T_{min} and T_{max}, which occurs at $t = 0$ (or $t = \tau$) and $t = \kappa\tau$ respectively:

$$T_{max} = k_2 + T_2 \quad \text{and} \quad T_{min} = k_1 + T_1 \tag{10}$$

The average temperature T_{avg} and the ripple Υ can then be computed with:

$$\begin{aligned} T_{avg} &= \frac{\Delta_T}{2} \frac{1 - e^{-b\kappa\tau} - e^{-b\tau} + e^{-b(1-\kappa)\tau}}{1 - e^{-b\tau}} + T_2 \simeq \Delta_T \kappa + T_2 \\ \Upsilon &= \Delta_T \frac{1 - e^{-b\kappa\tau} - e^{-b(1-\kappa)\tau} + e^{-b\tau}}{1 - e^{-b\tau}} \simeq b\Delta_T \kappa(1 - \kappa)\tau \end{aligned} \tag{11}$$

Note that these expressions depend on the duty ratio and

- the higher κ is, the higher the average steady-state temperature is. Also, one can verify the equilibrium temperatures T_1 and T_2 when the input power is always P_1 and P_2 respectively. They were initially defined in (7). They can as well be deduced from (11) when the duty ratio is $\kappa = 1$ and $\kappa = 0$ respectively;
- the ripple is maximum when $\kappa = 0.5$.

Furthermore, the temperature variation depends on the period: the smaller τ the lower the ripple.

Remark that chopping the power mode application has a nice effect on the maximum temperature T_{max} in (10) – as well as on the average temperature T_{avg} in (11) – as it is lower than the one achieved at V_{high}, i.e. $T_{high} = T_1$, as soon as the power (and consequently the voltage) oscillates, that is when the duty ratio κ is lower than 1.

Transient Phase Analysis. The behaviour of the temperature in the transient phase is now analysed. Suppose that the system runs at V_{low} for a long time, the temperature being stabilized in its lower value T_2. Then, a new task that needs the PE to (partially) run under V_{high}, is applied. Thus, the chopping scheme presented above is applied. At the very beginning of this chopping process, the temperature is not in steady-state, i.e. $T(t=0) \neq T(t=\tau)$. During the transient phase, the thermal expression in (6) becomes:

$$T(t) = \begin{cases} k_1(n)e^{-b(t-n\tau)} + T_1 & \text{if} \quad n\tau \leq t \leq n\tau + \kappa\tau \\ k_2(n)e^{-b(t-\kappa\tau-n\tau)} + T_2 & \text{if} \quad n\tau + \kappa\tau \leq t \leq (n+1)\tau \end{cases}$$

where $n \in \mathbb{N}$ is incremented at each new oscillating period. The parameters in (8) now depend on n:

$$\begin{aligned} k_1(n) &= -\varDelta_T \left[1 + \left(1 - e^{b\kappa\tau}\right) \sum_{i=1}^{n} e^{-ib\tau} \right] \\ k_2(n) &= \varDelta_T \left(1 - e^{-b\kappa\tau}\right) \sum_{i=0}^{n} e^{-ib\tau} \end{aligned} \tag{12}$$

The geometric series can be expressed as follows

$$\sum_{i=1}^{n} e^{-ib\tau} = e^{-b\tau} \frac{1 - e^{-nb\tau}}{1 - e^{-b\tau}}$$

As a consequence, the transient phase parameters can be given with their steady-state values – previously defined in (8) – plus a term decreasing with respect to n:

$$\begin{aligned} k_1(n) &= k_1 + \varDelta_T \frac{1 - e^{b\kappa\tau}}{1 - e^{-b\tau}} e^{-(n+1)b\tau} \\ k_2(n) &= k_2 - \varDelta_T \frac{1 - e^{-b\kappa\tau}}{1 - e^{-b\tau}} e^{-(n+1)b\tau} \end{aligned}$$

Note that the latter term exhibits an exponential decrease. Moreover, the initial temperature slope $\mu(n=0)$ can be expressed from (9) and (12). Then, at each new oscillating period, the slope is equal to the previous one plus an extra term η defined as follows:

$$\mu(n) = \mu(n-1) + \eta(n) \tag{13}$$
$$\text{with} \quad \eta(n) = \sigma b \Delta_T \Big(1 - e^{\sigma b \kappa \tau}\Big) e^{-nb\tau}$$
$$\text{and} \quad \sigma := \begin{cases} +1 & \text{if} \quad n\tau \leq t \leq n\tau + \kappa\tau \\ -1 & \text{if} \quad n\tau + \kappa\tau \leq t \leq (n+1)\tau \end{cases}$$

Remark that from (12) one can derive the steady-state expressions in (8), i.e. when $n \mapsto \infty$, using the polylogarithm closed form:

$$\sum_{i=1}^{\infty} e^{-ib\tau} = \frac{e^{-b\tau}}{1 - e^{-b\tau}}$$

The resulting "turn-on" transient phase is depicted in Figure 5(b) where the initial temperature is equal to T_2 defined in (7). An input power P_1 is then applied during the first oscillating period, with the switch in position 1. Hence, the temperature increases with an initial slope equal to $\mu(0) = b\Delta_T$. During the second part of the oscillating period (i.e. after the switch has changed from position 1 to position 2), the temperature decreases with an initial slope equal to $\mu(0) = -b\Delta_T\left(1 - e^{-b\kappa\tau}\right)$. At time $t = \tau$, a new oscillating period occurs and $n = 1$. The temperature during the first part of this second oscillating period increases, but with a smaller initial slope when compared with the first oscillating period since the added term η is negative, see (13). Then, the temperature decreases during the second part of the second oscillating period, but with a higher slope since the added term η is positive during the second subinterval. The process is repeated until the steady state condition is attained.

Length of the Oscillating Period τ. The length τ of the oscillating period has to be fixed by the designer. Firstly, it could be constant (as assumed until now). Otherwise, it could vary with respect to a given criterion to satisfy. As shown in (11), τ fixes the temperature ripple in steady state (which also varies with κ). A small period leads to small variations. However, fast oscillations increase the energy consumption because the transitions consume some power that is not strictly speaking used for the computations.

Hence, the energy-temperature tradeoff clearly appears here and a suggested solution is to have slow oscillations when the temperature is not "too" important, that is when the average temperature is low – i.e. κ is small, as suggested in (11) – and, inversely, to allow fast oscillations when the temperature becomes high, that is when the duty ratio is important. This results in fixing the product $\kappa\tau$ instead of choosing a constant oscillating period, such that

$$\tau = \frac{\varphi}{\kappa} \tag{14}$$

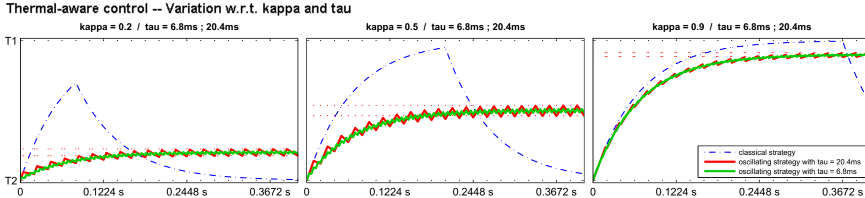
where φ denotes the *rising temperature time* when the switch is set in position 1. Note that this parameter has also to be chosen by the designer but the energy-temperature tradeoff is now self-adjusted.

Another solution could be to decide this time from the monitoring of the chip internal temperature. This latter solution has not been studied here. Actually, the proposed strategy is based on the previous work, summarised in subsection 4.1, whose low-cost implementation property was demonstrated in [23]. Moreover, taking into account the thermal constraints only requires to run a given task in a chopped version. This requires to determine the duty ratio and the oscillation period defined in (5) and (14) respectively, which are both easily computed. Moreover, the robustness property is still available. As a result, the thermal-aware proposal is low cost and robust to PVT variability.

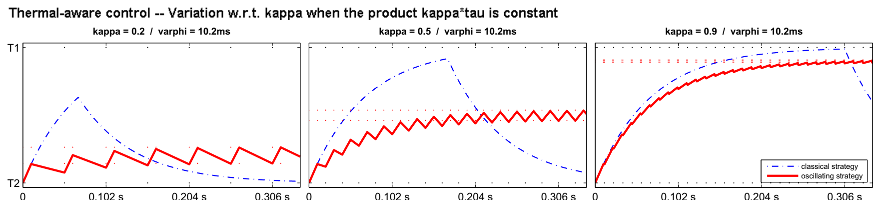
Simulation Results. It is quite difficult to find experimental values for the parameters of the thermal model (4) in the literature. For the simulation presented now, the ones deduced from [32][35][36] will be used. Thus, the different values are $R = 2\,°C/W$ and $C = 34\,mJ/°C$, which leads to a thermal time constant of $68\,ms$. The oscillating period τ is chosen smaller than the thermal time constant. Moreover, the ambient temperature is $T_{amb} = 25°C$ and the chip temperature varies from 35 to $95°C$ when successively applying a power of 5 and $35\,W$.
The different tests are depicted in Figure 6, where the chopped schemes as well as the classical strategy are represented. For the classical scheme, the task deadline is supposed equal to the time of the whole simulation, i.e. $\tau_{simu} = 0.4\,s$. The system then runs with the switch in position 1 during the time interval $\kappa\tau_{simu}$ and then with the switch in position 2 during $(1 - \kappa)\tau_{simu}$. As expected, the temperature highly increases.

Thanks to the chopped scheme for the DVFS application, the temperature can be reduced. Some tests are performed for different values of the duty ratio κ and of the oscillating period τ. The corresponding chip temperatures are depicted in Figure 6(a). As shown above, the steady-state average temperature and its ripple depend on the duty ratio κ and the length of the period τ. Consequently, the maximal attained temperature highly depends on the switching control since it could be decreased with a small duty ratio and some fast oscillations. However, a certain tradeoff between temperature and energy has to be taken into account. For this reason, it is proposed to automatically compute the length of the oscillating period from (14) in order to bound the time when the temperature increase is allowed. This solution is highlighted in Figure 6(b) where $\varphi = 0.3\,s$. Finally, one can remark slow and fast oscillations when the temperature is low and high respectively (i.e. when κ is small and large respectively).
The different cases presented highlight how hot spots within the chip can be reduced using the chopped strategy. Indeed, thanks to such a scheme both the temperature variations and the maximal temperature are decreased, when compared to the classical scheme.

The whole thermal-aware control of the energy-performance tradeoff is also tested in simulation. The idea is to reduce the energy consumption while ensuring "good" computational performance, as suggested in subsection 4.1, taking also into account the temperature within the chip. A scenario with three tasks to be executed is run, the number of instructions and deadline for each task



(a) Dynamics of the temperature with respect to the duty ratio κ and to the length of the oscillating period τ.



(b) Dynamics of the temperature when the product $\kappa\tau$ is constant, for different values of κ.

Fig. 6. Simulation results for the thermal-aware control scheme: impact of the different parameters on the average temperature and the temperature ripple

being known. The simulation results are depicted in Figure 7 where the product between the duty ratio and the period is fixed in order to manage the tradeoff between energy and temperature, i.e. $\varphi = 6.8\,ms$ in (14). The plots on the top part of Figure 7 show the number of instructions, the deadline, and the laxity. The speeds (average speed setpoint, predicted, measured) are provided on the plot in the middle of the figure, as well as the voltage value. The bottom plot shows the temperature of the chip, which is computed with (4). The temperature reached with the classical strategy (without any thermal management) is also represented. When the task to be run has a high computational load, the system runs during $\kappa\tau$ and $(1-\kappa)\tau$ at high and low voltage respectively. Then, this is repeated until the end of the task. As expected, the energy saving is (lightly) less important than in the classical strategy but the maximal temperature highly decreases (about 1 % increase and 40 % decrease respectively). This approach will basically decrease the hot spot appearance.

5 Conclusion and Future Work

In this paper, a thermal-aware DVFS approach is proposed. The scheme is based upon previously published DVFS strategies but the various power modes are applied, for each task to be run on the processing element not one after the other one, but in a chopped way. Moreover, instead of taking explicitly into account the temperature constraint, which leads to a complex optimisation problem, the temperature is smoothed as the main advantage of the "chopped-DVFS" is that

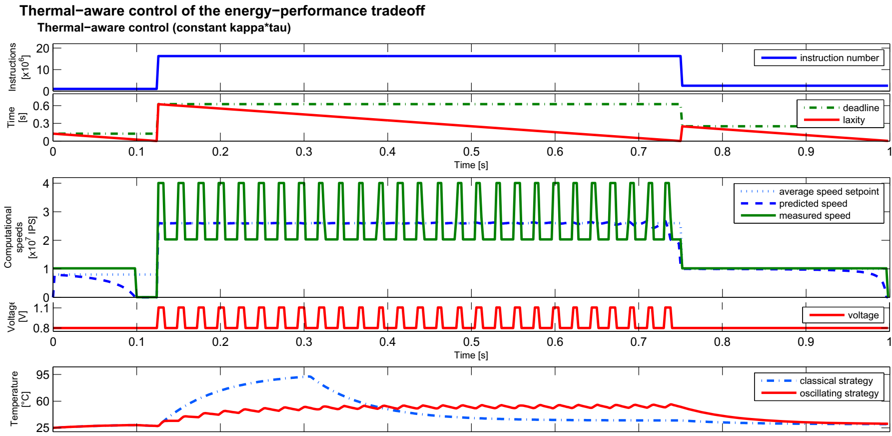


Fig. 7. Simulation results of the thermal-aware energy-performance tradeoff control with constant rising temperature time

it limits the temperature increase/decrease, with a small temperature ripple that depends on the oscillating period.

Some simulations results have been provided to show the effectiveness of the approach proposed and its capability to efficiently reduce the temperature of the chip. Note that the control law is based on a fast predictive control that has low computational cost. Thus it can be used at run-time, and executed if necessary at each oscillating period.

The chopped scheme proposed here will be implemented on the LoCoMoTIV Silicon platform [37][38][39] that has been designed at CEA-LETI Minatec Campus in order to evaluate advanced closed-loop control strategies for power management and PVT variability mitigations for advanced technology computing platforms.

Acknowledgements. This work has been performed while S. Durand was post-doc fellow in the joint NeCS Team, INRIA/GIPSA-lab. It has been conducted in cooperation with CEA, LETI Minatec Campus, under the PILSI-CRI, Grenoble, France. It has also received partial funding from the ARTEMIS Joint Undertaking under grant agreement number 100230 (SMECY project) and from the national funding authorities. It has been also supported by the FP7 project Pro3D under grant agreement number 278776.

References

1. Liu, Y., Yang, H., Dick, R.P., Wang, H., Shang, L.: Thermal vs energy optimization for dvfs-enabled processors in embedded systems. In: 8th International Symposium on Quality Electronic Design, ISQED 2007 (2007)

2. Hanumaiah, V., Vrudhula, S.: Temperature-aware dvfs for hard real-time applications on multi-core processors. IEEE Transactions on Computers (2011)
3. Durand, S.: Reduction of the Energy Consumption in Embedded Electronic Devices with Low Control Computational Cost. PhD thesis, University of Grenoble, France (2011)
4. Zhuravlev, S., Saez, J.C., Blagodurov, S., Fedorova, A., Prieto, V.: Survey of energy-cognizant scheduling techniques. IEEE Transactions on Parallel and Distributed Systems (2012)
5. Tschanz, J., Kao, J., Narendra, S., Nair, R., Antoniadis, D., Chandrakasan, A., De, V.: Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage. In: 2002 IEEE International Solid-State Circuits Conference, Digest of Technical Papers, ISSCC, vol. 1, pp. 422–478 (2002)
6. Firouzi, F., Yazdanbakhsh, A., Dorosti, H., Fakhraie, S.M.: Dynamic soft error hardening via joint body biasing and dynamic voltage scaling. In: 2011 14th Euromicro Conference on Digital System Design, DSD, August 31-September 2, pp. 385–392 (2011)
7. Mehta, N., Amrutur, B.: Dynamic supply and threshold voltage scaling for cmos digital circuits using in-situ power monitor. IEEE Transactions on Very Large Scale Integration (VLSI) Systems PP(99), 1–10 (2011)
8. Horowitz, M., Indermaur, T., González, R.: Low-power digital design. In: IEEE Symposium on Low Power Electronics, Digest of Technical Papers, pp. 8–11 (October 1994)
9. Choudhary, P., Marculescu, D.: Hardware based frequency/voltage control of voltage frequency island systems. In: Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2006, pp. 34–39 (October 2006)
10. Choi, K., Soma, R., Pedram, M.: Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 24(1), 18–28 (2005)
11. Sabry, M., Coskun, A., Atienza, D., Simunic Rosing, T., Brunschwiler, T.: Energy-efficient multi-objective thermal control for liquid-cooled 3D stacked architectures. IEEE Transactions on Computer Aided Design (2011)
12. Wang, X., Ma, K., Wang, Y.: Adaptive power control with online model estimation for chip multiprocessors. IEEE Trans. on Parallel and Distributed Systems 29(10), 1681–1696 (2011)
13. Chen, P., Chen, C.-C., Tsai, C.-C., Lu, W.-F.: A time-to-digital-converter-based cmos smart temperature sensor. IEEE Journal of Solid-State Circuits 40, 1642–1648 (2005)
14. Aoki, H., Ikeda, M., Asada, K.: On-chip voltage noise monitor for measuring voltage bounce in power supply lines using a digital tester. In: International Conference on Microelectronic Test Structures, ICMTS (2000)
15. Datta, B., Burleson, W.: Low-power and robust on-chip thermal sensing using differential ring oscillators. In: 50th Midwest Symposium on Circuits and Systems, MWSCAS (2007)
16. Quenot, G., Paris, N., Zavidovique, B.: A temperature and voltage measurement cell for vlsi circuits. In: Euro ASIC 1991 (1991)
17. Vincent, L., Beigne, E., Alacoque, L., Lesecq, S., Bour, C., Maurine, P.: A fully integrated 32 nm multiprobe for dynamic pvt measurements within complex digital soc. In: VARI 2011, Grenoble, France (2011)

18. Vincent, L., Maurine, P., Lesecq, S., Beigné, E.: Embedding statistical tests for on-chip dynamic voltage and temperature monitoring. In: 49th ACM/EDAC/IEEE Design Automation Conference, DAC (2012)
19. Chandrakasan, A.P., Brodersen, R.W.: Minimizing power consumption in digital CMOS circuits. Proceedings of the IEEE 83(4), 498–523 (1995)
20. Varma, A., Ganesh, B., Sen, M., Choudhury, S.R., Srinivasan, L., Bruce, J.: A control-theoretic approach to dynamic voltage scheduling. In: Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (2003)
21. Ishihara, T., Yasuura, H.: Voltage scheduling problem for dynamically variable voltage processors. In: Proceedings of the International Sympsonium on Low Power Electronics and Design (1998)
22. Miermont, S., Vivet, P., Renaudin, M.: A Power Supply Selector for Energy- and Area-Efficient Local Dynamic Voltage Scaling. In: Azémard, N., Svensson, L. (eds.) PATMOS 2007. LNCS, vol. 4644, pp. 556–565. Springer, Heidelberg (2007)
23. Durand, S., Marchand, N.: Fully discrete control scheme of the energy-performance tradeoff in embedded electronic devices. In: Proceedings of the 18th World Congress of IFAC (2011)
24. Alamir, M.: Stabilization of Nonlinear Systems Using Receding-Horizon Control Schemes: A Parametrized Approach for Fast Systems. LNCIS, vol. 339. Springer, Heidelberg (2006)
25. Kuzmicz, W., Piwowarska, E., Pfitzner, A., Kasprowicz, D.: Static power consumption in nano-cmos circuits: Physics and modelling. In: Proceeding of the 14th International Conference on Mixed Design of Integrated Circuits and Systems (2007)
26. Yang, C.-Y., Chen, J.-J., Lothar, T., Kuo, T.-W.: Energy-efficient real-time task scheduling with temperature-dependent leakage. In: Conference & Exhibition on Design, Automation and Test in Europe (2010)
27. Yuan, L., Leventhal, S., Qu, G.: Temperature-aware leakage minimization technique for real-time systems. In: IEEE/ACM International Conference on Computer-Aided Design (2006)
28. Chaturvedi, V., Huang, H., Quan, G.: Leakage aware scheduling on maximum temperature minimization for periodic hard real-time systems. In: 10th IEEE International Conference on Computer and Information Technology (2010)
29. Huang, H., Quan, G.: Leakage aware energy minimization for real-time systems under the maximum temperature constraint. In: Conference & Exhibition on Design, Automation and Test in Europe (2011)
30. Chaturvedi, V., Quan, G.: Leakage conscious DVS scheduling for peak temperature minimization. In: 16th Asia and South Pacific Design Automation Conference (2011)
31. Huang, H., Quan, G., Fan, J., Qiu, M.: Throughput maximizaion for periodic real-time systems under the maximal temperature constraint. In: 48th ACM/EDAC/IEEE Design Automation Conference (2011)
32. Quan, G., Zhang, Y.: Leakage aware feasibility analisys for temperature-constrained hard real-time periodic tasks. In: 21st Euromicro Conference on Real-Time Systems (2009)
33. Zhang, S., Chatha, K.S.: Approximation algorithm for the temperature-aware scheduling problem. In: IEEE/ACM International Conference on Computer-Aided Design (2007)
34. Erickson, R.W., Maksimović, D.: Fundamentals of Power Electronics, 2nd edn. Springer Science (2001)

35. Sridhar, A., Vincenzi, A., Ruggiero, M., Brunschwiler, T., Atienza, D.: 3D-ICE: Fast compact transient thermal modeling for 3D-ICs with inter-tier liquid cooling. In: International Conference on Computer-Aided Design (2010)
36. Sridhar, A., Vincenzi, A., Ruggiero, M., Brunschwiler, T., Atienza, D.: Compact transient thermal model for 3D ICs with liquid cooling via enhanced heat transfer cavity geometries. In: 16th International Workshop on Thermal Investigations of ICs and Systems (2010)
37. Beigne, E., Vivet, P.: An innovative local adaptive voltage scaling architecture for on-chip variability compensation. In: IEEE Int. Conf. New Circuits and Systems, NEWCAS, pp. 510–513 (June 2011)
38. Albea, C., Puschini, D., Vivet, P., Miro Panades, I., Beigné, E., Lesecq, S.: Architecture and robust control of a digital frequency-locked loop for fine-grain dynamic voltage and frequency scaling in globally asynchronous locally synchronous structures. J. Low Power Electronics 7(3), 328–340 (2011)
39. STMicroelectronics and CEA. Platform 2012 – A Manycore Programmable Accelerator for Ultra-Efficient Embedded Computing in Nanometer Technology (November 2010) (Whitepaper)

Author Index

Aldinucci, Marco 199, 218, 257
Atienza, David 277

Bacivarov, Iuliana 277, 294
Basu, Ananda 277, 314
Beigné, Edith 334
Benini, Luca 277
Bennaceur, Amel 133
Bensalem, Saddek 277, 314
Bourgos, Paraskevas 314
Bozga, Marius 277, 314
Breddin, Tino 237
Breu, Ruth 150, 169
Brown, Christopher 218, 237
Bruni, Roberto 49

Campa, Sonia 257
Cesarini, Francesco 218
Chokshi, Devesh B. 277

Danelutto, Marco 199, 218, 237
de Leastar, Eamonn 184
De Micheli, Giovanni 277
De Nicola, Rocco 25
Durand, Sylvain 334

Elger, Peter 184

Fabre, Christian 277, 334
Felderer, Michael 150, 169
Ferrari, Gianluigi 25
Flamand, Éric 277

Gander, Matthias 150, 169
González-Vélez, Horacio 218
Griffin, Leigh 184

Hähnle, Reiner 109
Hammond, Kevin 218, 237
Helvensteijn, Michiel 109
Hölzl, Matthias 1

Issarny, Valérie 133

Johansson, Richard 133
Johnsen, Einar Broch 89, 109

Katt, Basel 150, 169
Keller, Rainer 218
Kilpatrick, Peter 199, 218, 237, 257
Krimm, Jean-Pierre 277
Kumar, Pratyush 277

Lamprecht, Anna-Lena 69
Leblebici, Yusuf 277
Lesecq, Suzanne 334
Lienhardt, Michael 109
Loreti, Michele 25

Maheshwari, Mayur 314
Margaria, Tiziana 69
Melgratti, Hernán 49
Melpignano, Diego 277
Montanari, Ugo 49
Montangero, Carlo 199
Moschitti, Alessandro 133
Mottin, Julien 277

Pugliese, Rosario 25
Puschini, Diego 334

Rossbory, Michael 218
Ruggiero, Martino 277

Sabry, Mohamed M. 277
Sangiorgi, Davide 109
Schaefer, Ina 69, 109
Schlatte, Rudolf 89
Schöner, Holger 237
Schor, Lars 277, 294
Semini, Laura 199
Shainer, Gilad 218
Sifakis, Joseph 314
Spalazzese, Romina 133
Steffen, Bernhard 69
Sykes, Daniel 133

Tapia Tarifa, Silvia Lizeth 89
Thiele, Lothar 277, 294
Tordini, Fabio 257
Torquati, Massimo 257
Tribastone, Mirco 1

Vincent, Lionel 334

Wirsing, Martin 1
Wong, Peter Y.H. 109

Yang, Hoeseok 277, 294

Zambonelli, Franco 1