Thomas Gschwind
Flavio De Paoli
Volker Gruhn
Matthias Book (Eds.)

LNCS 7306

Software Composition

11th International Conference, SC 2012
Prague, Czech Republic, May/June 2012
Proceedings

Springer

Lecture Notes in Computer Science 7306

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

Thomas Gschwind Flavio De Paoli
Volker Gruhn Matthias Book (Eds.)

Software Composition

11th International Conference, SC 2012
Prague, Czech Republic, May 31 – June 1, 2012
Proceedings

Springer

Volume Editors

Thomas Gschwind
IBM Zurich Resarch Lab
Säumerstrasse 4
8803 Rüschlikon, Switzerland
E-mail: thg@zurich.ibm.com

Flavio De Paoli
University of Milano - Bicocca
Department of Informatics, Systems and Communication
Viale Sarca 336/14
20126 Milano, Italy
E-mail: depaoli@disco.unimib.it

Volker Gruhn
Matthias Book
University Duisburg-Essen
paluno – The Ruhr Institute for Software Technology
Gerlingstraße 16
45127 Essen, Germany
E-mail: {volker.gruhn; matthias.book@paluno.uni-due.de}

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-30563-4 e-ISBN 978-3-642-30564-1
DOI 10.1007/978-3-642-30564-1
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012937847

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

LNCS Sublibrary: SL 2 – Programming and Software Engineering

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

The 11th International Conference on Software Composition (SC 2012) provided researchers and practitioners with a unique platform to present and discuss challenges of how composition of software parts may be used to build and maintain large software systems. Co-located with the TOOLS 2012 Federated Conferences in Prague, SC 2012 built upon a history of a successful series of conferences on software composition held since 2002 in cities across Europe.

We received 42 submissions co-authored by researchers, practitioners, and academics from over 20 countries. Each paper was peer-reviewed by at least three reviewers, and discussed by the Program Committee. Based on the recommendations and discussions, we accepted 12 papers, leading to an acceptance rate of 28.6%.

Besides these technical papers, we are excited to have won Uwe Assmann and Mehdi Jazayeri as keynote speakers for SC 2012, who shared their insights on aspects of software composition with the combined SC 2012 and TOOLS 2012 audience.

We are grateful to the members of the Program Committee and the external reviewers for helping us to seek submissions and provide valuable and timely reviews. Their efforts enabled us to put together a high-quality technical program for SC 2012. We are indebted to the local arrangements team of TOOLS 2012 for the successful organization of all conference and social events. The SC 2012 submission, review, and proceedings process was extensively supported by the EasyChair Conference Management System. We also acknowledge the prompt and professional support from Springer, who published these proceedings in printed and electronic volumes as part of the *Lecture Notes in Computer Science* series. Finally, we would like to thank our sponsors adesso AG and AOSD Europe for their generous support of this conference.

Most importantly, we would like to thank all authors and participants of SC 2012 for their insightful works and discussions!

May 2012

Thomas Gschwind
Flavio De Paoli
Volker Gruhn
Matthias Book

Organization

Program Committee

Sven Apel	University of Passau, Germany
Walter Binder	University of Lugano, Switzerland
Nikolaj Bjorner	Microsoft Research, USA
Eric Bodden	Technical University of Darmstadt, Germany
Jan Bosch	Chalmers University of Technology, Sweden
Siobhán Clarke	Trinity College Dublin, Ireland
Flavio De Paoli	University of Milano-Bicocca, Italy
Robert France	Colorado State University, USA
Harald Gall	University of Zurich, Switzerland
Jeff Gray	University of Alabama, USA
Volker Gruhn	University of Duisburg-Essen, Germany
Thomas Gschwind	IBM Research, Switzerland
Ethan Jackson	Microsoft Research, USA
Welf Löwe	Linnaeus University, Sweden
Raffaela Mirandola	Politecnico di Milano, Italy
Johann Oberleitner	bwin, Austria
Claus Pahl	Dublin City University, Ireland
Cesare Pautasso	University of Lugano, Switzerland
Florian Rosenberg	IBM T.J. Watson Research Center, USA
Jean-Guy Schneider	Swinburne University of Technology, Australia
Clemens Szyperski	Microsoft Research, USA
Salvador Trujillo	IKERLAN Research Centre, Spain
Eric Wohlstadter	University of British Columbia, Canada

Steering Committee

Sven Apel	University of Passau, Germany
Uwe Aßmann	Technical University of Dresden, Germany
Judith Bishop	Microsoft Research, USA
Thomas Gschwind	IBM Research, Switzerland
Oscar Nierstrasz	University of Bern, Switzerland
Mario Südholt	INRIA – Ecole des Mines de Nantes, France

Additional Reviewers

Ansaloni, Danilo
Bieman, Jim
Bouza, Amancio
Cho, Hyun
De Carlos, Xabier
De Sosa, Josune
Ghezzi, Giacomo
Groba, Christin
Hajebi, Saeed
Hert, Matthias
Hine, Cameron
Jacob, Ferosh
Kuhlemann, Martin
Murguzur, Aitor
Müller, Sebastian
O'Toole, Eamonn
Rodriguez, Diego
Sarimbekov, Aibek
Schulze, Sandro
Sinschek, Jan
Song, Hui
von Rhein, Alexander
Wijesiriwardana, Chaman
Yiu, Jian
Yokokawa, Akira

Sponsors

Keynote Speakers

Mehdi Jazayeri is professor of computer science and founding dean of the Faculty of Informatics at the University of Lugano. From 1994 through 2007, he was also professor of computer science and head of the Distributed Systems Group at the Technical University of Vienna. He is interested in programming, software engineering, programming languages, and distributed systems. He has worked at both technical and management capacities at Hewlett-Packard Laboratories, Palo Alto, Synapse Computer Corporation, Ridge Computers, and TRW Vidar. He spent two years in Pisa, Italy, to set up and manage a joint research project on parallel systems between Hewlett-Packard and the University of Pisa. He has been an assistant professor of computer science at the University of North Carolina at Chapel Hill, adjunct professor at Georgia Institute of Technology, University of Santa Clara, and San Jose State University. He was a Fulbright Scholar at the University of Helsinki (1979) and a visiting professor at the Politecnico di Milano (1988). He was a principal investigator on several European projects dealing with software architectures and advanced distributed systems.

Mehdi Jazayeri was named an IEEE Fellow in 2007. He is also a Member of ACM, the Austrian, German, and Swiss Computer Societies. He holds degrees from Massachusetts Institute of Technology (SB, 1971) and Case Western Reserve University (MS, 1973; PhD, 1975). He has been a consultant to the US Government and to multinational companies in the areas of software engineering, design, architecture, and processes.

Uwe Aßmann holds the Chair of Software Engineering at the Technical University of Dresden. He got a PhD in compiler optimization and a habilitation from Karlsruhe University on "invasive software composition" (ISC), a composition technology for code fragments enabling flexible software reuse. ISC unifies generic, connector-, view-, and aspect-based programming for arbitrary program or modeling languages. The technology is demonstrated by the Reuseware environment, a meta-environment for the generation of software tools (http://www.reuseware.org).

Currently, in the project "Highly Adaptive Energy-Efficient Computing (HAEC)" at TU Dresden, Uwe Aßmann's group applies ISC to energy-aware autotuning (EAT), a technique to dynamically recompose code adapted to the required quality of service, to the context of the system, and to the hardware platforms. EAT is based on multi-objective optimization (MOO) and always delivers an optimal system configuration with respect to the context parameters. It is a promising technology also for the optimization of other qualities of future cyber-physical systems (CPS).

Table of Contents

Software Composition in Specification Languages

Context-Aware and Dynamic Composition

Composition in Software Development

Applications of Software Composition

Modeling Dynamic Architectures Using Dy-BIP*

Marius Bozga, Mohamad Jaber, Nikolaos Maris, and Joseph Sifakis

UJF-Grenoble 1 / CNRS, VERIMAG UMR 5104, Grenoble, F-38041, France
Firstname.Lastname@imag.fr

Abstract. Dynamic architectures in which interactions between components can evolve during execution, are essential for modern computing systems such as web-based systems, reconfigurable middleware, wireless sensor networks and fault-tolerant systems. Currently, we lack rigorous frameworks for their modeling, development and implementation. We propose Dy-BIP a dynamic extension of the BIP component framework rooted in rigorous operational semantics and supporting a powerful and high-level set of primitives for describing dynamic interactions. These are expressed as symbolic constraints offered by interacting components and computed efficiently by an execution Engine. We present experimental results which validate the effectiveness of Dy-BIP and show significant advantages over using static architecture models.

1 Introduction

Architectures are essential for mastering the complexity of systems and facilitate their analysis and evolution. They allow a separation between detailed behavior of components and their overall coordination. Coordination is usually expressed by constraints that define possible interactions between components. There exists a large number of formalisms supporting a concept of architecture, including software component frameworks, systems description languages and hardware description languages. Despite an abundant literature and a considerable volume of research, there is no agreement on a common concept of architecture, while most definitions agree on the core e.g. diagrammatic representations by using connectors. This is due to two main reasons.

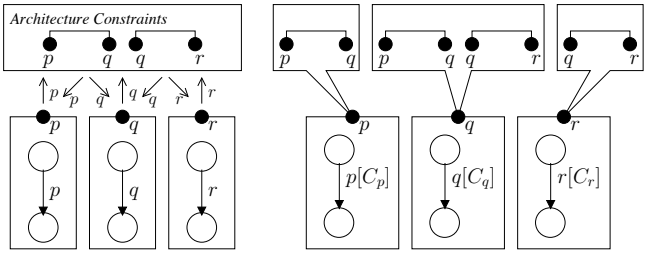
First, is the lack of rigorous operational semantics defining architectures as composition operators on components. That is the behavior of a composite component is inferred from the behavior of its constituent components by applying architectural constraints. For existing component frameworks, the definition of rigorous operational semantics runs into many technical difficulties. They fail to clearly separate between behavior of components and architecture. Connectors are not just memoryless switching elements. They can be considered as special types of components with memory e.g. fifo queues and specific behavior. Another difficulty stems from verbose architecture definitions e.g. by using ADLs [1], that do not rely on a minimal set of concepts. Such definitions are hardly amenable to formalization. Finally, some frameworks [2] use declarative languages e.g. first order logic to express global architecture constraints which are useful for checking correctness but as a rule do not provide a basis for defining operational semantics.

* This work is partially supported by the FP7 IP ASCENS.

T. Gschwind et al. (Eds.): SC 2012, LNCS 7306, pp. 1–16, 2012.

The second reason is the distinction between static and dynamic architectures. Usually, hardware and system description languages rely on static architecture models. The relationships between components are known at design time and are explicitly specified as a set of connectors defining possible interactions. Dynamic architectures are needed for modeling reconfigurable systems or systems that adapt their behavior to changing environments. They are defined as the composition of dynamically changing architecture constraints offered by their constituent components. Filling the gap between static and dynamic architecture models raises a set of interesting problems. In principle, dynamic architecture models are more general: each configuration corresponds to a static architecture model. Is it possible to define a dynamic architecture modeling language as an extension of a static architecture modeling language? Furthermore, if we restrict to systems with a finite - although potentially large - set of possible configurations, any dynamic architecture model can be translated into a static architecture model. Such a translation can yield very complex static architecture models. As a rule, using dynamic architectures may lead to more concise models. However, static architecture models can be executed more efficiently thanks to the global and static knowledge of connectors [3].

We propose the Dy-BIP component framework based on rigorous operational semantics for modeling both static and dynamic architectures. Dy-BIP can be considered as an extension of the BIP language [4] for the construction of composite hierarchically structured components from atomic components. These are characterized by their behavior specified as automata extended with data and functions described in C. A transition of an automaton is labeled by a port name, a guard (boolean condition on local data) and an action (computation on local data). In BIP architectures are composition operators on components defining their interactions. An interaction is described as a set of ports from different components. It can be executed if there exists a set of enabled transitions labeled by its ports. The completion of an interaction is followed by the completion of the involved transitions: execution of the corresponding actions followed by a move to the target state. An operational semantics for BIP has been defined in [5]. It provides a basis for the implementation of an Engine that orchestrates component execution. The Engine knows the set of the interactions modeling the architecture. It executes cyclically and atomically the following three-step protocol: 1) from a state each component sends to the Engine the ports of its enabled transitions; 2) the Engine computes the set of feasible interactions (sets of received ports corresponding to some interaction); 3) the Engine chooses non-deterministically one interaction amongst the feasible interactions by sending back to the components the names of their ports involved in this interaction. Figure 1(a) shows a static architecture defined by interactions pq and qr. Its consists of three components offering communications through ports p, q and r. In contrast to BIP, the set of interactions characterizing architectures in Dy-BIP changes dynamically with states. A port p has an associated architecture constraint C_p which describes possible sets of interactions involving p. Feasible interactions from a state are computed as maximal solutions of constraints obtained as the conjunction of constraints offered by enabled transitions. Figure 1(b) illustrates a dynamic architecture with three components offering ports p, q and r with associated constraints C_p, C_q and C_r. As for the static architecture, the possible interactions are pq and qr.



(a) Static architecture (b) Dynamic architecture

Fig. 1. Static and dynamic architecture

We provide operational semantics for Dy-BIP implemented by an Engine which as for BIP, orchestrates components by executing atomically a three-step protocol. The protocol differs in that components send not only port names of enabled transitions but also their associated architecture constraints.

Dy-BIP is an extension of BIP. A BIP model with a global architecture constraint C, can be represented as a Dy-BIP model such that the constraint C_p associated with a port p is the set of the interactions of C involving p.

Dy-BIP allows modeling dynamic architectures as the composition of instances of component types. For the sake of simplicity, we assume that there is no dynamic creation/deletion of component instances. The main contributions are the following:

- Definition of a logic for the description of architecture constraints. The logic is not only expressive but also amenable to analysis and execution. It encompasses quantification over instances of component types. Formulas involve port names used as logical variables and characterise sets of interactions. Given a formula, a feasible interaction is any set of ports assigned true by a valuation which satisfies the formula.
- Study of a semantic model and a modeling methodology for writing architecture constraints associated with ports. For a port p, the associated constraint is decomposed into three types of constraints characterizing interaction between ports [6]: "causal constraint", "acceptance constraint", "filter constraint". The causal constraint defines the ports required for interaction. The acceptance constraint defines optional ports for participation. The filter constraint is an invariant used to discriminate undesirable configurations of a component's environment.
- Implementation principles for Engines handling symbolic architecture constraints. The proposed implementation is based on the resolution of architecture constraints on-the-fly. The Engines use efficient constraint resolution techniques based on BDDs. For a given model, quantifiers over components can be eliminated and formulas become boolean expressions on ports.
- Experimental results and benchmarks showing benefits from using dynamic architectures compared to static architectures. We consider several examples showing that compositional modeling of dynamic architectures allows enhanced conciseness and rigorousness. In particular, it is possible to master complexity of intricate dynamic interactions by compositional specification of interactions of individual components.

The paper is structured as follows. Section 2 presents related work. Section 3 describes the semantic model for Dy-BIP. Section 4 presents the dynamic architecture description language and the methodology for writing constraints. Section 5 is dedicated to examples and experimental results. Section 6 concludes and discusses future work directions.

2 Related Work

In contrast to other frameworks [7,8] Dy-BIP relies on a clear distinction between behavior and architecture as a set of stateless architecture constraints characterizing interactions. Architecture constraints are specified compositionally as the conjunction of individual architecture constraints of components. Existing frameworks usually describe dynamic architectures as a set of global transitions between configurations. Only process algebras adopt a compositional approach e.g. pi-calculus [9]. Nonetheless, they do not encompass a concept of architecture as behavior and composition operators are intermingled. Dy-BIP differs from other formalisms such as [10] in that it has rigorous operational semantics. In [2], a first order logic extended with architecture-specific predicates is used. However, there is no clear methodology on how to express synchronisation protocols (e.g., rendezvous, broadcast) whose combination is expressive enough to represent any kind of interaction and avoids the exhaustive enumeration of all possible interactions [11]. In [12], a dynamic architecture is defined as a set of global transitions between global configurations. These transitions are expressed in a first order logic extended with architecture-specific predicates. The same logic is used in [13,14] but global configurations are computed at runtime from the local constraints of each component. Dy-BIP follows the same approach but constraints are stateless (they are based on the boolean representation of causal rules [6]) and take advantage of the stateful behavior of the components by eliminating some of the undesirable global configurations implicitly. [15] provides an operational semantics based on the composition of global configurations from local ones. These express three forms of dependencies between services (mandatory, optional and negative). Nonetheless, dynamism is supported only at the installation phase.

In BIP [4], coordination between components is modeled by using connectors [6]. A simple (or *flat*) connector is an expression of the form $p'_1 \dots p'_k p_{k+1} \dots p_n$ where primed p'_i ports are triggers, and unprimed ports p_j are synchrons. For a flat connector involving the set of ports $\{p_1, \dots, p_n\}$, interaction semantics defines the set of its interactions γ by the following rule: an interaction is any non-empty subset of $\{p_1, \dots, p_n\}$ which contains some port that is trigger; otherwise (if all ports are synchrons), the only possible interaction is the maximal one, that is $p_1 \dots p_n$.

Connectors, representing these two protocols for a sender s and receivers r_1, r_2, r_3, are shown in 2-(a,b). Triangles represent triggers, whereas bullets represent synchrons. Hierarchical connectors are expressions composed of types ports and/or typed sub-connectors. Figure 2-c shows a connector realizing an atomic broadcast from a port s to ports r_1, r_2, r_3. The sending port s is trigger, and the three receiving ports are strongly synchronized in a sub-connector itself typed as a synchron. The connector shown in Figure 2-d is a causal chain of interactions initiated by the port s, the possible interactions are $s, sr_1, sr_1r_2, sr_1r_2r_3$.

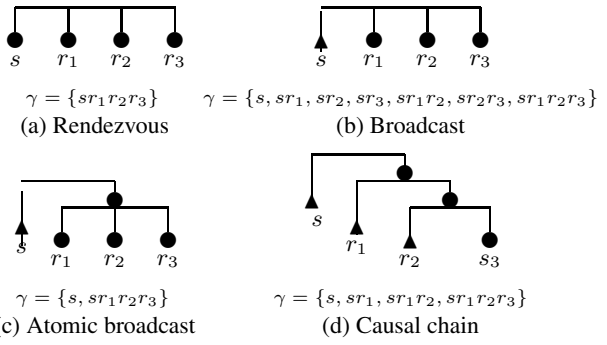


Fig. 2. Graphic representation of connectors

Connectors can be used to define easily any type of coordination between components. However, in this case connectors and their interactions are defined statically, and this leads to inefficiency for systems with dynamically changing architecture.

3 The Dynamic-BIP Model

Dy-BIP offers primitives for dynamic architecture modeling. Atomic components are transition systems. Transitions are labeled with ports, that is, action names, and constraints for interaction with other components.

Ports are used to define interactions between atomic components. Henceforth $\mathcal{P}$ is a universal set of ports. An interaction is a non empty subset $a \subseteq \mathcal{P}$. To simplify notation, an interaction $a = \{p_1, p_2, \ldots p_n\}$ is simply denoted by $a = p_1 p_2 \ldots p_n$.

3.1 Interaction Constraints

To introduce dynamic architectures, we consider that each atomic component provides its own interaction constraints at each computation step. A global interaction is defined as a solution of the set of interaction constraints offered by components. As the interactions at some state may depend on interactions in the past, it is necessary to parametrize interaction constraints by *history variables* which keep track of the interactions already executed. For example, in a protocol, if A sends a message to B, then A can record the identity of B in a history variable to remember that an acknowledgement is expected from B. If A does not use history variables then it is necessary to encode the identity of the receiver in control locations and this may result in an explosion of the number of its control locations.

Definition 1 (Interaction Constraint). *Given a set of history variables H, an interaction constraint C is defined by the following grammar:*

$$C ::= \texttt{true} \mid p \in h \mid p \mid \neg C \mid C \wedge C \tag{1}$$

where $p \in \mathcal{P}$ is a port, and $h \in H$ is a history variable.

The syntax of interactions constraints is tacitly extended for all boolean operators e.g., `false`, $\Rightarrow$, $\vee$ in the usual way. To simplify notation, we overload the meaning of a port symbol p: within $p \in h$ it denotes the port itself whereas p alone denotes a boolean variable. This ambiguity is removed in the two-step definition of semantics given below. We denote by $\mathcal{C}$ the set of all interaction constraints and by $\mathcal{C}^b$ the subset of boolean constraints, that is, without history variables.

Given a valuation of history variables $\mu : H \mapsto 2^{\mathcal{P}}$, an interaction constraint $C \in \mathcal{C}$ defines a boolean interaction constraint $[\![C]\!]_\mu$ by the following rules:

$$\begin{array}{ll} [\![\texttt{true}]\!]_\mu & = \texttt{true} \\ [\![p \in h]\!]_\mu & = \begin{cases} \texttt{true} & \text{if } p \in \mu(h) \\ \texttt{false} & \text{otherwise} \end{cases} \\ [\![p]\!]_\mu & = p \\ [\![\neg C]\!]_\mu & = \neg [\![C]\!]_\mu \\ [\![C_1 \wedge C_2]\!]_\mu & = [\![C_1]\!]_\mu \wedge [\![C_2]\!]_\mu \end{array} \tag{2}$$

That is, the terms of the form $p \in h$ are replaced by `true` or `false` as the case may be: `true` if port p belongs to the stored interaction and `false` otherwise. All the other terms of the interaction constraint remain unchanged.

An interaction $a \subseteq \mathcal{P}$ satisfies a boolean constraint $C \in \mathcal{C}^b$ (denoted by $a \models C$), as defined by the following rules:

$$\begin{array}{ll} a \models \texttt{true} & \\ a \models p & \Leftrightarrow p \in a \\ a \models \neg C & \Leftrightarrow \neg (a \models C) \\ a \models C_1 \wedge C_2 & \Leftrightarrow (a \models C_1) \wedge (a \models C_2) \end{array} \tag{3}$$

We denote by $\mathcal{I}(C)$ and $\mathcal{I}^{max}(C)$ the set of interactions and respectively maximal interactions satisfying C, formally:

$$\begin{array}{ll} \mathcal{I}(C) & = \{a \mid a \models C\} \\ \mathcal{I}^{max}(C) & = \{a \in \mathcal{I}(C) \mid \nexists a' \in \mathcal{I}(C) \,.\, a' \supset a\} \end{array} \tag{4}$$

For example, we can specify interaction constraints for: (1) *rendez-vous* between p_1, p_2, p_3 by $C_1 = (p_1 \Rightarrow p_2) \wedge (p_2 \Rightarrow p_3) \wedge (p_3 \Rightarrow p_1)$, the only possible interaction is the maximal one, that is $\mathcal{I}(C_1) = \mathcal{I}^{max}(C_1) = \{p_1p_2p_3\}$; (2) *broadcast* where s is the sending port and r_1, r_2, r_3 are the receiving ports by $C_2 = (true \Rightarrow s) \wedge (r_1 \Rightarrow s) \wedge (r_2 \Rightarrow s) \wedge (r_3 \Rightarrow s)$, the possible interactions are $\mathcal{I}(C_2) = \{s, sr_1, sr_1r_2, sr_1r_2r_3\}$ and $\mathcal{I}^{max}(C_2) = \{sr_1r_2r_3\}$; (3) the constraint $p \Rightarrow \texttt{false}$ means absence of p from any interaction; (4) $p \Rightarrow \texttt{true}$ allows inclusion of p in any interaction.

3.2 Atomic Components

An atomic component is an automaton extended with history variables. Transitions represent relations on control locations (local states). Each transition is labeled by a port, an interaction constraint and a set of history variables to be updated.

Definition 2 (Atomic Component). *An atomic component is a tuple* $B = (L, P, H, T)$*, where,*

- L *is a finite set of control locations;*
- $P \subseteq \mathcal{P}$ *is a finite set of ports;*
- H *is a finite set of history variables;*
- $T \subseteq L \times P \times \mathcal{C} \times 2^H \times L$ *is a finite set of transitions. Each transition* $(\ell, p, C, \mathbf{h}, \ell')$*, denoted by* $\ell \xrightarrow{p, C, \mathbf{h}} \ell'$ *is labeled with:*
  - $p \in P$*, the port offered for interaction*
  - $C \in \mathcal{C}$*, the interaction constraint on* $\mathcal{P}$ *and* H
  - $\mathbf{h} \subseteq H$*, the set of history variables to be updated*

Given $\mu : H \mapsto 2^{\mathcal{P}}$ a valuation of the history variables H, the state of an atomic component $B = (L, P, H, T)$ is a pair (ℓ, μ), where $\ell \in L$ is a control location. $Q = L \times \boldsymbol{\mu}$ is the set of states ot the atomic component B, where $\boldsymbol{\mu}$ denotes the set of valuations on H. For each state $(\ell, \mu) \in Q$, we define its associated *state constraint* $SC(\ell, \mu)$ as follows:

$$SC(\ell, \mu) = \left[\bigvee_{\ell \xrightarrow{p, C, \mathbf{h}} \ell'} \left(p \wedge [\![C]\!]_\mu \wedge \bigwedge_{p' \in P \setminus \{p\}} \neg p' \right) \right] \vee \bigwedge_{p \in P} \neg p \tag{5}$$

The state constraint characterizes the set of possible contributions of the component to a global interaction at state (ℓ, μ). Either the component executes some transition $\ell \xrightarrow{p, C, \mathbf{h}} \ell'$ and offers interactions (1) involving p and excluding all other ports labeling transitions from this state, that is, $p \wedge \bigwedge_{p' \in P \setminus \{p\}} \neg p'$ holds and (2) involving ports which satisfy the constraint C for the valuation μ, that is, $[\![C]\!]_\mu$. Or, the component does not interact, if none of its ports is used, that is $\bigwedge_{p \in P} \neg p$.

3.3 Composition

A system $S = B_1 \| \ldots \| B_n$ is defined as the composition of a set of atomic components $B_i = (L_i, P_i, H_i, T_i)_{i=1,n}$. We assume that the sets of locations, ports and history variables are pairwise disjoint. The semantics of the composition is provided by the following definition.

Definition 3 (Composition). *The behavior of a system* $S = B_1 \| \ldots \| B_n$ *is a labeled transition system* $(Q, \Sigma, \rightarrow)$*, where,*

- $Q = \prod_{i=1}^{n} Q_i$ *is the set of states, where* Q_i *is the set of states of component* B_i*;*
- $\Sigma = 2^P$ *is the set of interactions, where* $P = \cup_{i=1}^{n} P_i$*;*
- $\rightarrow \subseteq Q \times \Sigma \times Q$ *is the set of transitions, defined by the following rule:*

$$\frac{\{\mu_i : H_i \mapsto P\}_{i=1}^n \quad a = \{p_j\}_{j\in J} \in \mathcal{I}^{max}(\bigwedge_{i=1}^n SC(\ell_i,\mu_i)) \quad \forall j \in J.\ \ \ell_j \xrightarrow{p_j, C_j, \mathbf{h}_j} \ell'_j \quad \bigwedge_{j\in J}[\![C_j]\!]_{\mu_j} \quad \mu'_j = \mu_j[a/\mathbf{h}_j] \quad \forall j \notin J.\ \ \ell'_j = \ell_j \quad \mu'_j = \mu_j}{((\ell_1,\mu_1),\ldots,(\ell_n,\mu_n)) \xrightarrow{a} ((\ell'_1,\mu'_1),\ldots,(\ell'_n,\mu'_n))}$$

Intuitively, system transitions are taken for maximal interactions a that satisfy *all* state constraints defined by atomic components $\bigwedge_{i=1}^n SC(\ell_i,\mu_i)$. The components participating in the interaction change their location according to the selected transition and update their local valuation of history variables. The components which do not participate keep their state unchanged.

The composition semantics has been implemented by using a centralized execution Engine. The Engine gathers current state interaction constraints from all atomic components. Then, it builds the overall system of constraints and solves it, that is, finds the set of maximally satisfying interactions. One maximal interaction is then selected and executed atomically by all involved components. This operation is repeated by the Engine on-the-fly, at every state reached at execution. More formally, atomic components interact and coordinate their execution through the Engine according to the following protocol:

1. Each component B_i, computes its state constraint $SC(\ell_i,\mu_i)$ and sends it to the Engine.
2. The Engine computes the global constraint GC as the conjunction of state constraints defined by atomic components $GC = \bigwedge_{i=1}^n SC(\ell_i,\mu_i)$.
3. The Engine picks some maximal interaction a on $\mathcal{P}$, such that $a \models GC$. That is, $a \in \mathcal{I}^{max}(GC)$.
4. The Engine notifies the selected interaction a to all participating components, that is, all B_j such that $a \cap P_j \neq \emptyset$.
5. Each notified component B_j executes its local transition labeled by $p_j = a \cap P_j$ and updates its history variables.

Example 1. Figure 3 shows an example of a composite component consisting of three atomic components B_1, B_2, and B_3. Component B_1 requests synchronization with either B_2 or B_3, and then performs some computation with the synchronized component. To do so, we model component B_1 as follows. It has two control locations l_1, l_2, two ports s, r, and a history variable h. From control location l_1, the transition labeled by the port s requests synchronisation with ports b_1 or b_2 ($s \Rightarrow b_1 \vee b_2$), forbids the synchronisation with b_1 and b_2 at the same time ($\neg(b_1 \wedge b_2)$), and forbids the synchronisation with f_1 and f_2 ($\neg f_1 \wedge \neg f_2$). After executing this transition, the executed interaction is stored in the history variable h (**Update** h), to keep track of the identity of the synchronized component. From control location l_2, the transition labeled by the port r, depending on the value of the history variable h, either (1) requests synchronisation with port f_1 and forbids the synchronisation with port f_2; or (2) requests synchronisation with port f_2 and forbids the synchronisation with port f_1. In both cases, it forbids the synchronisation with ports b_1 and b_2 ($\neg b_1 \wedge \neg b_2$).

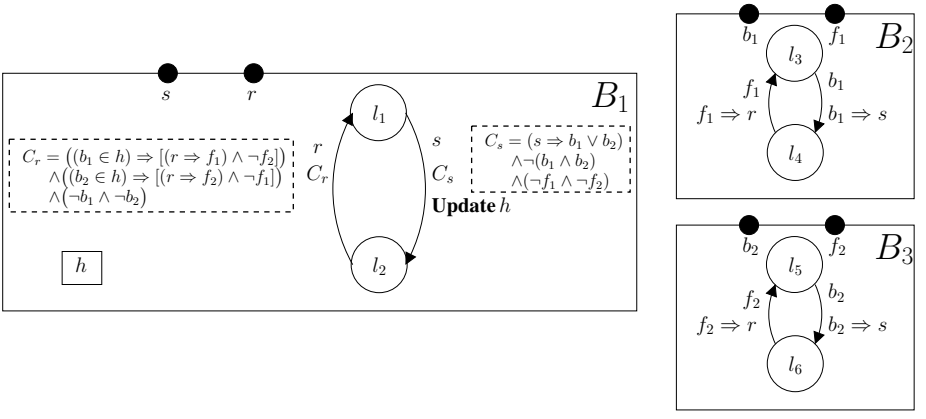


Fig. 3. An example of a composite component in Dy-BIP

From location l_1 the state constraint is $SC(l_1, \mu_1) = (s \wedge C_s \wedge \neg r) \vee (\neg s \wedge \neg r)$ regardless of the value of μ_1. This is due to the fact that constraint C_s does not depend on history variables. For this constraint, the possible interactions are $\mathcal{I}^{max}(SC(l_1, \mu)) = \{sb_1, sb_2\}$. From location l_2, if the history variable h contains the interaction sb_2, then the state constraint is $SC(l_2, \mu_1) = (r \wedge [\![C_r]\!]_{\mu_1} \wedge \neg s) \vee (\neg s \wedge \neg r)$, where $[\![C_r]\!]_{\mu_1} = \big(\texttt{false} \Rightarrow [(r \Rightarrow f_1) \wedge \neg f_2]\big) \wedge \big(\texttt{true} \Rightarrow [(r \Rightarrow f_2) \wedge \neg f_1]\big) \wedge \big(\neg b_1 \wedge \neg b_2\big)$. For this constraint, the only possible interaction is $\mathcal{I}^{max}(SC(l_2, \mu_1)) = \{rf_2\}$.

Initially, components B_1, B_2, and B_3 are in locations l_1, l_3, and l_5, respectively. Starting from these locations, the Engine coordinates execution of these components as follows: (1) Components B_1, B_2, and B_3 compute their state constraints $SC(l_1, \mu_1)$, $SC(l_3, \mu_2)$, and $SC(l_5, \mu_3)$, respectively, where, $SC(l_1, \mu_1) = (s \wedge C_s \wedge \neg r) \vee (\neg s \wedge \neg r)$, $SC(l_3, \mu_2) = (b_1 \wedge (b_1 \Rightarrow s) \wedge \neg f_1) \vee (\neg b_1 \wedge \neg f_1)$, and $SC(l_5, \mu_3) = (b_2 \wedge (b_2 \Rightarrow s) \wedge \neg f_2) \vee (\neg b_2 \wedge \neg f_2)$; (2) The Engine picks any interaction from $\mathcal{I}^{max}(SC(l_1, \mu_1) \wedge SC(l_3, \mu_2) \wedge SC(l_5, \mu_3)) = \{sb_1, sb_2\}$; (3) If the Engine selects the interaction sb_2, it notifies components B_1 and B_3; (4) Components B_1 and B_3 execute their transitions labeled by s and b_2, respectively. Moreover, component B_1 sets its history variable h to sb_2.

4 Methodology for Writing Interaction Constraints

Writing interaction constraints associated with transitions of atomic components, in the proposed declarative language may be error-prone or may lead to incomplete specifications. We provide a methodology based on a classification of constraints and on a set of macro-notations for enhancing soundness and completeness. The classification distinguishes between interactions in which a port p must, may or must not be involved. It allows a systematic analysis of interaction capabilities of components to make sure that no essential properties are omitted. Macro-notations allow a compact and high-level expression of the most commonly used constraints, thus avoiding specification errors.

We extend the interaction constraint language towards a first order logic with quantification over component instances. This extension is useful because in practice, systems are built from multiple, replicated instances of components of different types. The formulas of the logic interaction constraints are therefore defined as follows:

$$C ::= \texttt{true} \mid x.p \in h \mid x.p \mid x = y \mid x = \texttt{self} \mid \neg C \mid C \wedge C \mid \forall x{:}T.C(x) \qquad (6)$$

In this definition, T denotes a component type. Each component type represents a set of component instances with identical interfaces and behavior. The variables x, y range over component instances. These variables must occur in the scope of a quantifier, e.g. $\forall x : T.C(x)$. They are strongly typed and moreover, they can be tested for equality. Additionally, `self` represents a fixed component instance, that is, the (context) component where the constraint belongs. The remaining syntactic constructs are directly lifted from the propositional case: $h \in H$ denotes a history variable and $x.p$ denote the port p belonging to component instance x. As previously, we consider the standard extension of this logic for all boolean operators and existential quantification.

In the sequel, we consider systems consisting of finitely many instances for each component type. Under this restriction, an interaction constraint written in the logic above boils down into a propositional interaction constraint by (1) substituting `self` by the current component instance, (2) elimination of universal quantifiers and (3) evaluation of equality constraints. For example, for a component type T with instances $t_1, \ldots t_k$, universal quantifiers of the form $\forall x{:}T$ can be eliminated:

$$\forall x{:}T.C(x) \equiv C(t_1) \wedge \ldots \wedge C(t_k) \qquad (7)$$

In the rest of this section, we provide guidelines for writing interaction constraints in this logic. Consider a fixed transition in some component (type) which is labeled by a port p. The associated interaction constraint C_p is a conjunction of three types of constraints, respectively *causal*, *acceptance* and *filtering*, as explained below.

4.1 Causal Constraints

These constraints are used to specify the ports required for interactions of p. At propositional level, they can be reduced to the one of the two following forms, either $\texttt{true} \Rightarrow p$ or $p \Rightarrow C$, where C is a boolean interaction constraint without negated ports. To express such constraints in practice, we provide hereafter few useful abbreviations:

- **Trigger** $\equiv \texttt{true} \Rightarrow \texttt{self}.p$. This constraint specifies that port p is a trigger, that is, the transition labeled by **Trigger** does not require synchronization with transitions of other components to occur.
- **Require** $T.q \equiv \exists x : T.(\texttt{self}.p \Rightarrow x.q)$. This constraint specifies that an arbitrary instance of component type T must participate with the port q in the interaction involving p.
- **Require** $x_1.q_1 ... x_n.q_n[x_1 : T_1 \ldots x_n : T_n | C_1(x_1, ..., x_n)] \equiv \exists x_1 : T_1 ... \exists x_n : T_n.\ \big(C_1(x_1, ..., x_n) \wedge \texttt{self}.p \Rightarrow (x_1.q_1 \wedge \ldots \wedge x_n.q_n)\big)$. This is the most general type of require constraint. It specifies that a set of component instances $x_1, \ldots x_n$ satisfying the constraint $C_1(x_1 \ldots x_n)$ must jointly participate with respectively ports $q_1, \ldots q_n$ in the interaction involving p. Usually, the constraint C_1 is used to check previous participation of $x_1 \ldots x_n$ in interactions recorded into history variables.

4.2 Acceptance Constraints

These constraints define optional ports for participation used to define the boundary of interactions. They are expressed by excluding explicitly from interactions all the ports that are not optional. At propositional level, they are of the form $r \Rightarrow \texttt{false}$ where port r is excluded from interaction. In practice, we use the following abbreviations:

- **Accept** $T.q \equiv \bigwedge_{(T',q') \neq (T,q)} \forall x : T'.\big(x.q' \neq \texttt{self}.p \Rightarrow (x.q' \Rightarrow \texttt{false})\big)$. This constraint accepts only ports q of component instances of type T.
- **Accept** $x.q[x : T | x.r \in h] \equiv$ **Accept** $T.q \wedge \forall x : T.\big(x.r \notin h \Rightarrow (x.q \Rightarrow \texttt{false})\big)$. This constraint restricts participation to ports of component instances x that had participated in the interaction stored in h.

4.3 Filtering Constraints

These constraints are used to exclude some of interactions allowed by causal and acceptance constraints. At propositional level, filter constraints are of the form $p \Rightarrow C$ where each monomial in C has at least one negated port. In practice, we are commonly requiring unicity constraints of the form:

- **Unique** $T.q \equiv \forall x : T. \forall y : T.\ (x.q \wedge y.q \Rightarrow x = y)$. This constraint forbids the participation of p with more than one instance of component type T with the port q.

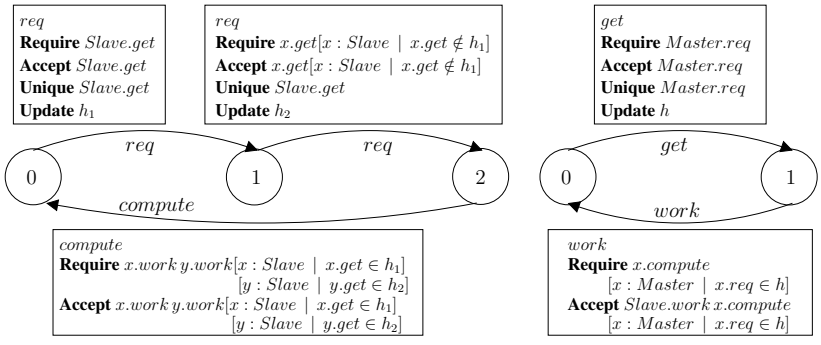
5 Experimental Results

The operational semantics of Dy-BIP has been implemented using the CUDD BDD package[1]. We compare execution times for Dy-BIP and BIP (where we define architecture statically) for a Master-Slave example. Two other non-trivial examples, Fault-Tolerant Servers, and Tracker and Peers are provided. Static architecture modeling for these examples in BIP leads to complex descriptions and may be error-prone. Dy-BIP models are concise and can be efficiently implemented. We present below the three examples and the simulation results. Execution times are provided for executing 1000 interactions of the Engine running on PC Quad-Xeon 2.67HGz with 6GB RAM.

5.1 Masters and Slaves

In this example, we consider two scenarios. In the first scenario, involves a system consisting of M Masters and S Slaves. Each Master sends requests sequentially to two Slaves, and then performs some computation involving both of them. The model of the Master component type is shown in Figure 4(a). Initially, from location 0, a Master requests synchronization with some Slave. To do so, it must synchronize his $request$ port with the get port of any Slave (**Require** $Slave.get$). As it requires one and only one Slave, we add a **Unique** constraint. When a Master instance m_1 synchronizes with

[1] CUDD: CU decision diagram package
(`http://vlsi.colorado.edu/~fabio/CUDD/`)



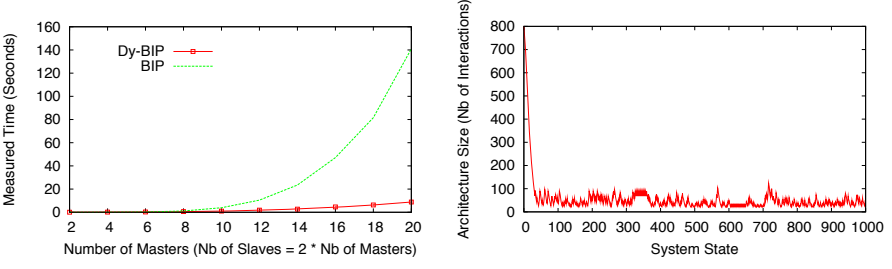
(a) Master type (b) Slave type

Fig. 4. Masters and Slaves in Dy-BIP

some Slave instance, say s_1, interaction $m_1.req\, s_1.get$ is chosen. Then, the Master m_1 has to keep track of the identity of the synchronized Slave (s_1) by saving the interaction $m_1.req\, s_1.get$ to the history variable h_1 (**Update** h_1). From location 1 the Master requires another Slave to synchronize with it, and keeps again track of its identity. Finally, from location 2, the Master establishes a ternary interaction with the two Slaves recorded in its history variables.

The Slave component type is shown if Figure 4(b). It accepts a request and provides a response. In order to allow participation in ternary interactions, the *Accept* clause includes the port of one Master and the work port of another Slave. Notice that, the statically predefined connector structure (using BIP) needs $M \times S + M \times S \times (S-1)$ interactions.

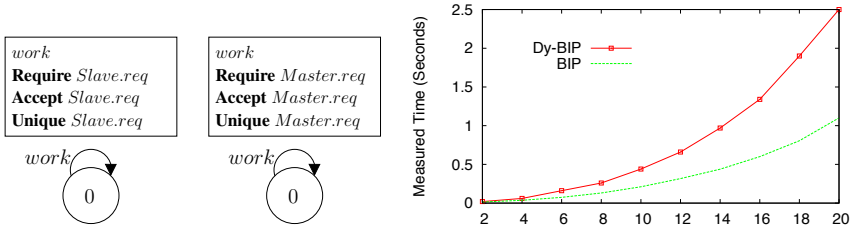
Figure 5(a) shows execution times of 1000 interactions for BIP and Dy-BIP, as a function of the number of components in the system. Dy-BIP considerably outperforms BIP. Figure 5(b) shows the number of the interactions created dynamically for Dy-BIP along the evolution of the system for 20 Masters and 40 Slaves. Notice that, for BIP the number of the interactions created is 32000 regardless of the system state.



(a) Master-Slave execution time (b) Architecture variation Dy-BIP

Fig. 5. Masters and Slaves execution time and variation of dynamic architecture

In the second scenario, we developed a simplified version of the Master-Slave model presented above. Each Master performs some computation in tight synchronization with any Slave. The model of the Master and Slave component types are shown in Figures 6(a) and 6(b). As expected, Figure 6(c) shows that the execution time for BIP outperforms Dy-BIP. This is due to the fact that the number of the interactions created is $M \times N$ for BIP as well as for Dy-BIP regardless of the system state.



(a) Master type (b) Slave type (c) Simple Master-Slave execution time

Fig. 6. Simple Master-Slave in Dy-BIP

5.2 Fault-Tolerant Servers

This example is inspired from the Fault-Tolerant Client-Server System presented in [3]. We consider a system consisting of a fixed number of available Servers and a pool of alternative Servers used in case of crash. When a Server crashes, another Server is turned on to preserve availability of the service, and so on. After successive crashes, we get a chain of Servers $s_1 \dots s_t$ where $s_1 \dots s_{t-1}$ are crashed and s_t is available. Crashes in this example are software crashes like a memory error. So, turning off and on a crashed Server is also a way of repairing it through an action called *softrepair*. Additionally, whenever a crashed Server s_i gets repaired, then the running Server in the chain should turn off s_t and all crashed Servers that replaced the repaired Server $(s_j, i+1 < j < t)$ provide a *softrepair*. Figure 5.2 shows a possible scenario with four Servers: (1) the first Server is turned on while all the others are turned off; (2) the first Server crashes and turns on the second Server; (3) the second Server crashes and turns on the third Server; (4) the first Server gets repaired. In this case, the third Server must be turned off and the crashed Server (the second one) has to turn off by

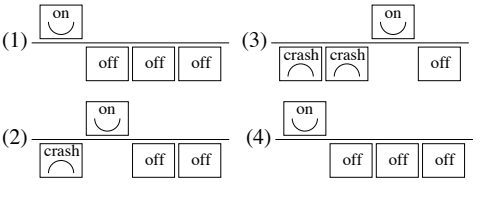


Fig. 7. Possible scenario for Fault-Tolerant Client-Server

executing *softrepair*. Notice that, the decision of turning off a running Server has to be propagated from the repaired Server. Also, Servers have to change state synchronously to keep constant the number of available Servers.

The model of a Server component is shown in Figure 8. Moreover, we use the plus symbol "+" to denote logical disjunction of require constraints involving several ports determined by the same expression. For instance, **Require** $x.repair{+}x.softrepair[x : Server \mid x.crash \in h_2] \equiv$ **Require** $x.repair[x : Server \mid x.crash \in h_2] \vee$ **Require** $x.softrepair[x : Server \mid x.crash \in h_2]$. A turnoff requires a repair or a *softrepair* from any Server that has crashed.

Notice that, statically predefined connector structure (using BIP) leads to more than N^N interactions, where N is the total number of Servers. For this reason, modeling statically this architecture is practically impossible even for a relatively small number of Servers. Figure 9(a) depicts execution time for Dy-BIP. For a total number of 27 Servers with 13 available Servers, executing 1000 interactions requires only 8 seconds. Notice that, using BIP we have to statically define more than 27^{27} interactions!

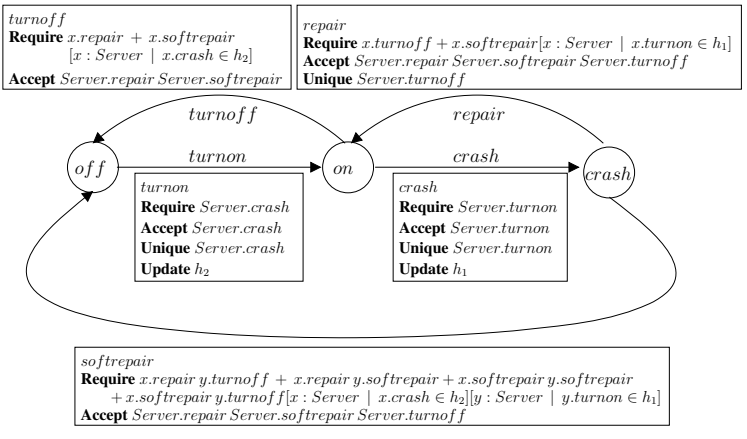
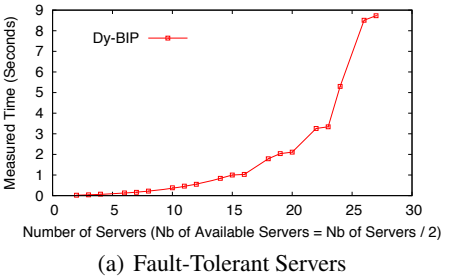
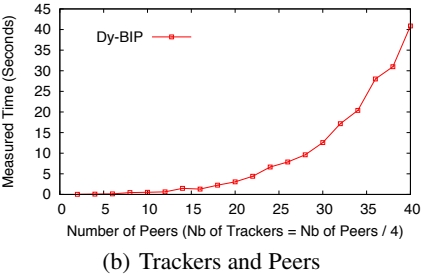


Fig. 8. Fault-Tolerant Servers in Dy-BIP



(a) Fault-Tolerant Servers



(b) Trackers and Peers

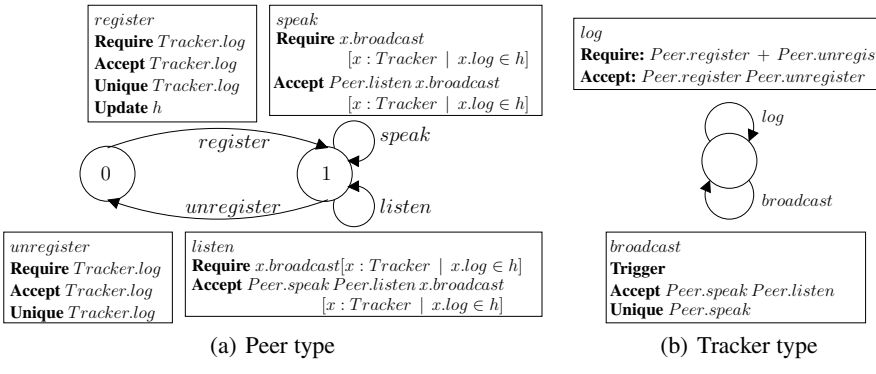
Fig. 9. Execution times for Fault-Tolerant Servers and Trackers and Peers

5.3 Trackers and Peers

As third benchmark, we consider a simplified wireless audio protocol for reliable multicast communication. The protocol allows an arbitrary numbers of participants (named *Peers*) to dynamically connect and communicate along an arbitrary number of wireless communication channels (managed by dedicated *Trackers*).

In this protocol, Peers are allowed to use at most one communication channel at a time. The access to channels is subject to an explicit registration mechanism. Dynamically, every Peer selects and registers to the channel it wants to use. Once registered, Peers can either speak, that is, send data over the channel, or listen, that is, receive data sent by others. For every channel, its associated Tracker ensures for any communication that (1) exactly one registered peer (on that channel) is speaking and (2) all other registered peers are listening. That is, data is delivered atomically to all potential receivers. Finally, Peers can dynamically de-register, then register to other channels, etc.

The Dy-BIP model is depicted in Figure 10. As for the previous example, we use the "+" abbreviation for disjunction of require constraints applied to several ports. Notice that, a statically predefined connector structure is exponentially complex and the model explodes rapidly. We need $P \times T \times (3 + 2^{P-1})$ connectors, where P is the total number of Peers and T is the total number of Trackers.



(a) Peer type (b) Tracker type

Fig. 10. Trackers and Peers in Dy-BIP

Figure 9(b) shows execution times for 1000 interactions for a Dy-BIP model where there are four times more Peers than Trackers.

6 Conclusion

The paper provides a simple modeling language for compositional description of dynamic architectures. The language as a straightforward extension of a static architecture modeling language, bridges the gap between static and dynamic description styles. It is simple but expressive enough as illustrated by non-trivial examples. Its associated modeling methodology leads to concise and intelligible models obtained as the composition

of components. Global architecture constraints can be synthesized by composing architecture constraints of individual components. Using history variables in components avoids state explosion and duplication of ports.

This work contrasts with existing approaches for dynamic architectures which lack clear semantics and are not compositional. It proposes a rigorous methodology for writing architecture constraints of components. In particular, the distinction between different types of constraints provides guidance for their soundness and completeness. The choice between different implementations permits exploration of efficiency trade-offs. On-the-fly computation seems more adequate for architectures with a large number of configurations while regular BIP execution is more advantageous for systems with a small number of configurations.

Further developments will focus on integrating data transfer, priorities, as well as creation and deletion of components.

References

1. Medvidovic, N., Taylor, R.N.: A Framework for Classifying and Comparing Architecture Description Languages. In: Jazayeri, M. (ed.) ESEC 1997 and ESEC-FSE 1997. LNCS, vol. 1301, pp. 60–76. Springer, Heidelberg (1997)
2. Garlan, D., Monroe, R.T., Wile, D.: Acme: An architecture description interchange language. In: Proceedings of CASCON 1997, Toronto, Ontario, pp. 169–183 (November 1997)
3. Allen, R.B., Douence, R., Garlan, D.: Specifying and Analyzing Dynamic Software Architectures. In: Astesiano, E. (ed.) ETAPS 1998 and FASE 1998. LNCS, vol. 1382, pp. 21–37. Springer, Heidelberg (1998)
4. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous component-based system design using the bip framework. IEEE Software 28(3), 41–48 (2011)
5. Basu, A., Bidinger, P., Bozga, M., Sifakis, J.: Distributed Semantics and Implementation for Systems with Interaction and Priority. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) FORTE 2008. LNCS, vol. 5048, pp. 116–133. Springer, Heidelberg (2008)
6. Bliudze, S., Sifakis, J.: Causal Semantics for the Algebra of Connectors (Extended Abstruct). In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 179–199. Springer, Heidelberg (2008)
7. Inverardi, P., Wolf, A.L.: Formal specification and analysis of software architectures using the chemical abstract machine model. IEEE Trans. Software Eng. 21(4), 373–386 (1995)
8. Métayer, D.L.: Describing software architecture styles using graph grammars. IEEE Trans. Software Eng. 24(7), 521–533 (1998)
9. Milner, R.: Communicating and mobile systems - the Pi-calculus. Cambridge University Press (1999)
10. Magee, J., Kramer, J.: Dynamic structure in software architectures. SIGSOFT Softw. Eng. Notes 21, 3–14 (1996)
11. Bliudze, S., Sifakis, J.: The algebra of connectors: structuring interaction in bip. In: EMSOFT, pp. 11–20 (2007)
12. Kim, J.S., Garlan, D.: Analyzing architectural styles. Journal of Systems and Software 83(7), 1216–1235 (2010)
13. Kacem, M.H., Jmaiel, M., Kacem, A.H., Drira, K.: Evaluation and comparison of adl based approaches for the description of dynamic of software architectures. In: ICEIS (3), pp. 189–195 (2005)
14. Georgiadis, I., Magee, J., Kramer, J.: Self-organising software architectures for distributed systems. In: WOSS, pp. 33–38 (2002)
15. Belguidoum, M., Dagnat, F.: Dependency management in software component deployment. Electr. Notes Theor. Comput. Sci. 182, 17–32 (2007)

Defining Composition Operators for BPMN*

Paul Istoan

CRP Gabriel Lippmann, Luxembourg, LASSY, University of Luxembourg, Luxembourg, Université de Rennes 1, France
istoan@lippmann.lu

Abstract. To support the constant evolution of modern enterprises, business process models are becoming exponentially more complex. Business process composition is regarded as a flexible mechanism capable to cope with this issue and make processes easier to understand, maintain and evolve. Composition operators are the foundation and the enablers of process composition. BPMN, the standard language for modelling business processes, is lacking such composition operators. Therefore, in this paper, we propose to extend the BPMN standard with a set of composition operators and with the concept of composition interface.

Keywords: Model composition, composition operators, Petri Nets, BPMN.

1 Introduction

The increasing transparency and accountability of all organisations, together with the modern complexity and importance of information and communications technology, tends to heighten the demand for process improvement. Business Process Modelling and Management (BPM) [1], [12] is an essential part of understanding and restructuring the activities an enterprise uses to achieve its business goals as efficiently as possible.

Modern enterprises are constantly changing and evolving [6]. Implicitly, business processes have to meet changes in application requirements, technology, policies and organization. As a result, they are becoming exponentially more complex: increased number of elements, high number of paths, processes model complex interactions between multiple actors and systems. This complexity makes a business process more difficult to understand and use when communicating with stakeholders. It also makes it complicated to determine if the process properly captures the right business practices and to validate it. Due to complexity, business processes also become more difficult to maintain and evolve over time, and at higher costs. Modelling complex, real-life business process models is of the utmost importance and represents a major challenge.

This situation emphasizes the need for separation of concerns (SOC) mechanisms [17] as support for modelling complex business processes. In the SOC

* This work has been funded by the SPLIT project (FNR + CNRS, FNR/INTER/CNRS/08/02).

T. Gschwind et al. (Eds.): SC 2012, LNCS 7306, pp. 17–34, 2012.

paradigm, concerns are defined separately and assembled into a final system using *composition operators* [23]. *Business process composition* [9] is regarded as a flexible mechanism capable to cope with the increasing complexity of business processes. Similar to component-based software development [13], the core idea is to create a complex process by assembling simpler ones. Process composition reduces complexity by having smaller process components connected together by flexible mechanisms to realize a process that provides the same business support as the initial complex process. The complexity of building a business process is taken away from the business analyst and delegated to the composition.

Creating a process by composition facilitates its understanding and its use. Moreover, it can be updated more easily, as the necessary changes are performed on smaller separate models. The maintainability of the business process is also enhanced. Another strong argument motivating the use of process composition is *process reuse* [16]. The desire to better manage processes and improve business efficiency has led to increased awareness and desire to reuse processes [7]. Process reuse is a way to promote the efficiency and quality of process modelling. Fewer business processes are built from scratch, as many existing processes are used for the development of new ones, following a compositional approach.

The general approach when applying model composition is to provide *composition operators*. They are mechanisms that take two (or more) models as input and generate an output that is their composition. Most languages provide a fixed set of composition operators, with explicit notations, specific behaviour and defined semantics. In case a language does not provide a composition operator with the desired behaviour, different workarounds need to be used.

In recent years, the Business Process Model and Notation 2.0 (BPMN) [2] has received increasing attention, becoming the standard for business processes modelling. We study how a complex BPMN process can be obtained through the composition of simpler ones. A thorough analysis of the BPMN specification [20] reveals that the standard does not address in any way nor does it provide support for business process composition. However, there are several possible workarounds. Conversations and choreographies, used to model interactions and message exchanges between participants, are a possible solution. Gateways, normally used to express control flow, can be used together with sub-processes, global tasks or call activities as a possible way to express process composition. Sub-processes are flow objects used as an abstraction mechanism in BPMN. They are used to hide or reveal additional levels of business process detail. Therefore, they can be used for hierarchical process decomposition. The use of sub-processes is a possible workaround for replacing some composition operators, like refinement. Nevertheless, complex compositions like choice or synchronization cannot be expressed using sub-processes. All these workarounds are very limited in terms of possible results that can be obtained. Composition of BPMN models currently requires specific knowledge in advance and takes up a lot of time and effort [18].

There are no *composition operators* available for BPMN. They are necessary to achieve the composition of BPMN processes. Therefore, the major contribution of this paper is to extend the BPMN standard with a set of composition

operators. To successfully apply a composition operator we must know where a business process can be connected with other processes. These are the places where the actual composition is performed. As a second contribution, we further extend BPMN with the notion of *composition interface* and explain how it facilitates the application of composition operators.

The standard defines the execution semantics of BPMN it in terms of enabling and firing of elements, based on a token-game. The start event generates a token The token traverses the sequence flow and passes through the flow objects of the process and it is eventually consumed at an end event. The behaviour of the business process can be described by tracking the path(s) of the token through the process. The dynamic behaviour of Petri Nets is also defined in terms of firing of transitions which triggers the passing of a token through the net. As the execution semantics of both languages are defined in a similar manner, we consider that Petri nets might provide useful composition operators that can also be applied to BPMN. Therefore, in this paper, Petri net composition operators are taken as the basis for constructing BPMN composition operators.

The rest of this paper is structured as follows. Section 2 briefly describes Petri nets and explains their relation with BPMN. An overview of some existing Petri net composition operators is also presented. Section 3 contains the main contribution of the paper. We provide a formalization of the BPMN abstract syntax and extend it with the *composition interface* concept. Then we define a set of composition operators for BPMN. In Section 4 we exemplify how a complex BPMN process can be constructed by applying several of the composition operators proposed. Finally, we draw some conclusions and present some perspectives for future work in Section 5.

2 Research Context

2.1 Petri Nets

The execution semantics of BPMN is defined in terms of enabling and firing of elements, based on a token-game. The dynamic behaviour of Petri Nets is also defined in terms of firing of transitions, triggering the passing of a token through the net. Due to this similarity of execution semantics, we consider appropriate to analyse closer the field of Petri nets and see if the composition operators defined for it can be a source of inspiration for defining BPMN composition operators.

Overview of Petri Nets. Petri nets [19] are a language for the modelling and validation of systems in which concurrency, communication and synchronisation play a major role [15]. It combines a rich mathematical theory with a useful graphical notation.

A Petri net [22] is a bipartite graphs containing *places* and *transitions* connected by directed *arcs*. Places model local system states, while transitions symbolise system actions. State changes are modelled by the *flow relation* which connects places with transitions and transitions with places using *arcs*. States of the system are represented by *markings*. A marking is a distribution of *tokens*

in the net. Tokens are denoted by a solid dot and can be put inside places. The initial state of the system is represented by the *initial marking*.

The semantics of a Petri net is described by the flow of tokens through the net. The token flow is initiated by the *firing of transitions*. First of all, a transition must be enabled. This happens when all of its input places contain at least one token. When enabled, transitions can be fired. When this happens, tokens are removed from all its input places and inserted into all its output places. The number of tokens removed and inserted may be different. The dynamic behaviour of a net is therefore described by a sequence of steps, where each step is the firing of a transition, taking the net from one marking to another.

Petri Net Composition Operators. Literature [4] [5] suggests a lot of different composition operators, for different purposes and different classes of nets. We briefly present in the following some of the best known operators.

The *sequential composition operator* [5] is the most commonly found. The result of applying it on two input nets S_1 and S_2 is a composite net that performs net S_1 first, followed by net S_2, in sequence, one after the other. In the result obtained, net S_1 must be completed before net S_2 can start. Most frequently, it is used when there is a causality relation, either logical or functional, between the two nets that are composed.

The *parallel composition operator* [11] defines the concurrent execution of two nets, independently of each other. Two nets can be executed concurrently if they do not depend on each other, i.e. they are not causally linked. Some authors [8] consider parallel composition as simply the disjoint union of two nets.

The *exclusive choice operator* [10] represents different possible paths of execution when the flow of control is determined based on a specific condition or decision, or even non-deterministically. It defines an alternate functionality, so the main goal of the net can be achieved in two (or more) distinct ways. Once one net executes its fist operation, the other net cannot be reached any more.

When applying the *choice composition operator* [3] on two input nets S_1 and S_2, we obtain a net that can perform either net S_1, either net S_2 or both of them. This means the nets are executed alternatively.

The *refinement composition operation* [25] models the transformation of a design from a high level abstract form to a lower level more concrete form. It consists in replacing a *place* or a *transition* in a net by another, more refined net, to introduce a higher level of detail. This operation is also known as *place refinement* or *transition refinement*.

The *synchronization composition operator* [21] specifies a situation in which two nets synchronize their execution because they have specific similarities between one or more places or transitions. The synchronization can be done both at the level of places (*place fusion*) or transitions (*transition synchronization*). For performing the actual synchronization operation, a matching process is performed to determines the *synchronization set*, which contains places and/or transitions from the two nets where the actual synchronization is performed. The result of this operator is a net that performs in parallel the parts of the two nets

that do not belong to the synchronization set, and merges (synchronizes) and performs only once the elements from the synchronization set.

2.2 BPMN: Business Process Modelling and Notation

The Business Process Modelling and Notation [2] is the standard for modelling business process flows proposed by the Object Management Group (OMG) [20]. Its primary goal is to provide a notation that is easily understandable by all business users.

The main concept defined in BPMN 2.0 is the *business process diagram (BPD)*, used to create graphical models of business processes and their operations. It is based on a flowchart technique. BPMN is easy to use and understand while also offering the expressiveness to model very complex business processes. A BPD is made up of several graphical elements, chosen to be distinguishable from each other and to utilize shapes that are familiar to most modellers [24].

There are four basic categories of elements defined: *Flow Objects*, *Connecting Objects*, *Swimlanes* and *Artifacts*. Flow Objects are the main graphical elements used to create BPDs and are separated into: *Events* (start, intermediate and end), *Activities* (atomic Task and compound Sub-Process) and *Gateways* (decision making, forking and merging of paths). Flow Objects are connected together to create the basic skeletal structure of a business process. These connectors are: *Sequence Flow*, *Message Flow* and *Association*. *Swimlanes* are used to group activities into separate categories for different functional capabilities or responsibilities. A *Pool* represents a participant in a process, and can be divided into *Lanes*. BPMN provides the ability to add context appropriate to a specific modelling situation. Any number of *Artifacts* can be added to a diagram as appropriate for the context of the business processes being modelled. Three types of artifacts are pre-defined: *Data Object*, *Group* and *Annotation*.

3 Creating BPMN Composition Operators

We start by introducing a set-based formalization of the abstract syntax of BPMN. This is required for defining the composition operators. We then introduce *composition interfaces* as an extension to BPMN and discuss their purpose. Finally, we define a set of composition operators for BPMN.

3.1 Abstract Syntax of BPMN Diagrams

As the BPMN standard does not provide a formal definition, we propose in the following a set-based formalization of the BPMN abstract syntax.

Definition 1. *We define the following notations for BPMN processes:*
Let $\mathcal{O}$ *be the set of all objects that appear in all BPMN diagrams.*
Let $\mathcal{F}$ *be the set of flow objects for all BPMN diagrams:* $\mathcal{F} \subseteq \mathcal{O}$
Let $\mathcal{A}$ *be the set of activities for all BPMN diagrams:* $\mathcal{A} \subseteq \mathcal{F}$
Let $\mathcal{E}$ *be the set of events for all BPMN diagrams:* $\mathcal{E} \subseteq \mathcal{F}$

Let $\mathcal{G}$ be the set of gateways for all BPMN diagrams: $\mathcal{G} \subseteq \mathcal{F}$

The set of flow objects is partitioned into disjoint sets of activities $\mathcal{A}$, events $\mathcal{E}$, and gateways $\mathcal{G}$: $\mathcal{F} = \mathcal{A} \cup \mathcal{E} \cup \mathcal{G}$

The set of events is partitioned into disjoint sets of start $\mathcal{E}_s$, intermediate $\mathcal{E}_i$, and end events $\mathcal{E}_e$: $\mathcal{E} = \mathcal{E}_s \cup \mathcal{E}_i \cup \mathcal{E}_e$

The set of gateways is partitioned into disjoint sets of parallel $\mathcal{G}_p$, exclusive $\mathcal{G}_x$, inclusive $\mathcal{G}_i$, and complex gateways $\mathcal{G}_c$: $\mathcal{G} = \mathcal{G}_p \cup \mathcal{G}_x \cup \mathcal{G}_i \cup \mathcal{G}_c$

The set of activities is partitioned into disjoint sets of tasks $\mathcal{T}$ and sub-processes $\mathcal{SP}$: $\mathcal{A} = \mathcal{T} \cup \mathcal{SP}$

Let $\mathcal{A}r$ be the set of artifacts for all BPMN diagrams: $\mathcal{A}r \subseteq \mathcal{O}$

Let $\mathcal{S}$ be the set of swimlanes for all BPMN diagrams: $\mathcal{S} \subseteq \mathcal{O}$

We propose the following definition for a BPMN process:

Definition 2. *A BPMN process is a tuple BP = (F, S, Ar, SF, MF, AS) where:*

F is the set of flow objects of the BPMN process, with $F \subseteq \mathcal{F}$

F is partitioned into disjoint sets of activities A, events E, and gateways G: $F = A \cup E \cup G$, where $A \subseteq \mathcal{A}, E \subseteq \mathcal{E}, G \subseteq \mathcal{G}$,

S is the set of swimlanes of the BPMN process, with $S \subseteq \mathcal{S}$

Ar is the set of artifacts of the BPMN process, with $Ar \subseteq \mathcal{A}r$

$SF \subseteq F \times F$ defines a sequence flow relation between flow objects

$MF \subseteq E \cup A \times E \cup A$ defines a message flow relation between events or activities

$AS \subseteq F \times Ar$ defines an association relation between flow objects and artifacts

We also define *predecessor* and *successor* functions for flow objects:

Let $pred : F \rightarrow F, pred(x) = \{y | (y, x) \in SF\}$

Let $succ : F \rightarrow F, succ(x) = \{y | (x, y) \in SF\}$

Additionally, we propose a set of consistency rules to assure the validity and correctness of the BPMN processes created. Due to lack of space, these rules are not presented here, but are available in [14].

3.2 Extending BPMN with Composition Interfaces

We extend the BPMN standard with the *composition tag* concept. A composition tag is a type of artifact and can be represented as a textual tag that is added on flow objects. It explicitly denotes the places where a business process can be composed with other ones. Composition tags can be added on any type of flow object (activity, event or gateway). A flow object having an input tag denotes that, by composition, another process fragment can be connected to the current one, right after the tagged flow object. Similarly, a flow object having an output tag denotes that another process fragment can be connected before the tagged flow object. We propose two types of composition tags: *input* and *output*.

$CT \subset \mathcal{A}r, CT = CT_i \cup CT_o$, partitioned into disjoint sets of input CT_i and output CT_o composition tags.

We also define a *tagging function* that returns the composition tag of a flow object, if it has one: $Tag : F \rightarrow CT$, where $F \subseteq \mathcal{F}$

To apply a composition operator on a BPMN process, we need to know the exact places where it can be composed with others. For this purpose, we introduce the notion of *composition interface*. A composition interface can be: *input* or *output*. They are defined using the previously introduced notion of composition tag. We define the *input composition interface* of a BPMN process as the set of all its flow objects tagged with an input composition tag, and the *output composition interface* as the set of all its flow objects tagged with and output composition tag.

Definition 3. *The composition interface of a BPMN process is* $I = I_i \cup I_o$*:*
I_i *is the input composition interface:* $I_i = \{x | x \in F, Tag(x) \in CT_i\}$
I_o *is the output composition interface:* $I_o = \{x | x \in F, Tag(x) \in CT_o\}$

We can now define the extended BPMN processes used as input for our composition operators:

Definition 4. *An extended BPMN process is a tuple* BP_{ext} *= (F, S, Ar, SF, MF, AS, Tag) where (F, S, Ar, SF, MF, AS) defines a normal business process and Tag is a tagging function returning the composition interfaces of flow objects.*

For an extended BPMN process, we define the following functions:
$out : BP \rightarrow E_e, out(BP) = \{e | Tag(e) \in CT_{in}\}, E_e = E \cap \mathcal{E}_e$ returns the end events of a process tagged with an input composition interface
$in : BP \rightarrow E_e, out(BP) = \{e | Tag(e) \in CT_{out}\}, E_s = E \cap \mathcal{E}_s$ returns the start events of a process tagged with an output composition interface

3.3 Sequential Composition Operator

To apply this operator, two conditions need to hold on the input BPMN models: BP_1 has an input composition interface at one of its end events; BP_2 has an output composition interface at its start event.

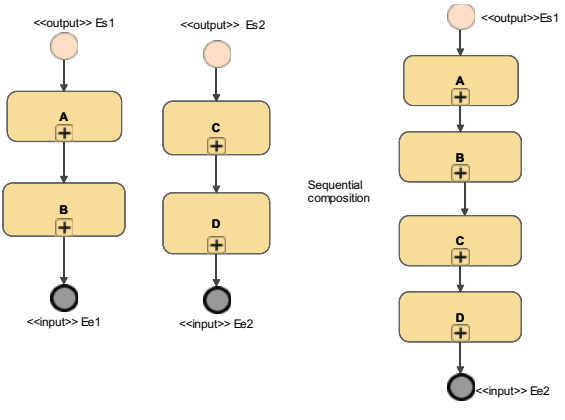


Fig. 1. Sequential composition operator for BPMN

Definition 5. *Let* $BP_1 = (F_1, S_1, Ar_1, SF_1, MF_1, AS_1, I_1)$ *and* $BP_2 = (F_2, S_2, Ar_2, SF_2, MF_2, AS_2, I_2)$ *be two business processes. The result of applying the sequential composition operator on processes* BP_1 *and* BP_2*, denoted* $seq(BP_1, BP_2)$*, is a new BPMN process* $BP_{res} = (F_{res}, S_{res}, Ar_{res}, SF_{res}, MF_{res}, AS_{res}, I_{res})$ *where:*

The flow objects of the result contains the union of the flow objects from the input models, from which we remove the end event of BP_1 *tagged with a composition interface and the start event of* BP_2*:*
$F_{res} = F_1 \cup F_2 \setminus \{out(BP1), in(BP2)\}$

For the resulting sequence flow, we need to disconnect out(BP1) and in(BP2) from the initial models, then connect together the remaining process fragments:
$SF_{res} = SF_1 \cup SF_2 \setminus \{(pred(out(BP1)), out(BP1)), (in(BP2), succ(in(BP2)))\} \cup (pred(out(BP1)), succ(in(BP2)))$

The swimlanes, artifacts, message flow and associations of the result are the union of their counterparts from the input processes:
$S_{res} = S_1 \cup S_2$, $Ar_{res} = Ar_1 \cup Ar_2$, $MF_{res} = MF_1 \cup MF_2$, $AS_{res} = AS_1 \cup AS_2$

The composition interface of the result is the union of the interfaces of the input models, from which we need to remove out(BP1) and in(BP2):
$I_{res} = I_{i-1} \setminus \{out(BP1)\} \cup I_{i-2} \setminus \{in(BP2)\}$

In terms of token passing, the semantics of this operator is the following: a new token is generated at the start event of BP_1 and through the outgoing sequence flow arrives at the first flow element of BP_1, enabling it. Once the flow element has executed, the token is sent through to the next flow element. In the same manner, the token traverses in sequence all the flow elements of BP_1, then those of BP_2, until it reaches the end event of BP_2 where it is consumed. As this is the only token generated, the process is considered completed. The trace obtained is the same as for the Petri net operator.

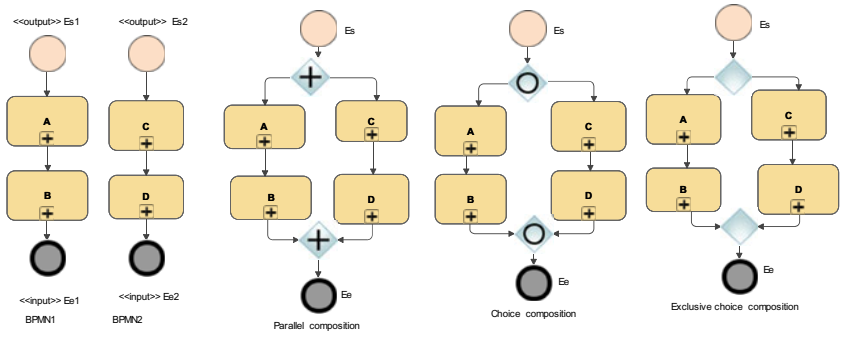


Fig. 2. Input BPMN models then parallel, choice and exclusive choice compositions

3.4 Parallel, Choice and Exclusive Choice Composition Operators

Although very different semantically, these three operators produce results that are similar from a structural point of view. They can be represented using the

same general pattern, with slight variations for each individual operator. Therefore, we introduce them in the following manner: the common parts are presented only once; we define precisely the aspects particular to each individual operator.

In order to apply any of these operators, two conditions need to hold on the input BPMN models: both $BPMN_1$ and $BPMN_2$ have an input composition interface at one of their end events and an output composition interface at their start events.

Definition 6. *Let* $BP_1 = (F_1, S_1, Ar_1, SF_1, MF_1, AS_1, I_1)$ *and* $BP_2 = (F_2, S_2, Ar_2, SF_2, MF_2, AS_2, I_2)$ *be two business processes. The result of applying the parallel/choice/exclusive choice composition operator on processes* BP_1 *and* BP_2*, denoted* $par(BP_1, BP_2)/cho(BP_1, BP_2)/exc(BP_1, BP_2)$ *is a new BPMN process* $BP_{res} = (F_{res}, S_{res}, Ar_{res}, SF_{res}, MF_{res}, AS_{res}, I_{res})$ *where:*

The result contains the union of all activities from the input processes: $A_{res} = A_1 \cup A_2$

The end result contains the union of the events of the input processes, from which we need to remove the start events of BP_1 *and* BP_2 *and their end events tagged with composition interfaces, then add a new start and end event:* $E_{res} = E_1 \cup E_2 \setminus \{in(BP1), out(BP1), in(BP2), out(BP2)\} \cup \{start_{new}, end_{new}\}$*, where* $E_1 \subseteq F_1, E_2 \subseteq F_2, start_{new} \in \mathcal{E}_s, end_{new} \in \mathcal{E}_e$

To obtain the gateways of the result, we take the union of the gateways of the input models and add two new gateways, different for each composition operator: $G_{res} = G_1 \cup G_2 \cup \{g_1, g_2\}$*, where* $G_1 \subseteq F_1, G_2 \subseteq F_2$ *and* $g_1, g_2 \in \mathcal{G}_p$ *for the parallel operator,* $g_1, g_2 \in \mathcal{G}_x$ *for the exclusive choice operator,* $g_1, g_2 \in \mathcal{G}_i$ *for the choice operator.*

For all these operators, the sequence flow is obtained from the union of sequence flows of the input models, from which we first need to disconnect the start and end events, then connect the new start and end events to the newly introduced gateways and finally connect these gateways to the remaining parts of the input processes:

$SF_{res} = SF_1 \cup SF_2 \setminus \{(in(BP1), succ(in(BP1))), (pred(out(BP1)), out(BP1)), (in(BP2), succ(in(BP2))), (pred(out(BP2)), out(BP2))\} \cup \{(start_{new}, g_1), (g_2, end_{new}), (g_1, succ(in(BP1))), (g_1, succ(in(BP2))), (pred(out(BP2)), g_2), (pred(out(BP1)), g_2)\}$

The swimlanes, artifacts, message flow and associations of the result are the union of their counterparts from the input processes:

$S_{res} = S_1 \cup S_2$, $Ar_{res} = Ar_1 \cup Ar_2$, $MF_{res} = MF_1 \cup MF_2$, $AS_{res} = AS_1 \cup AS_2$

The composition interface of the result contains the union of interfaces of the input models, from which we remove the start events and end events tagged with interfaces of BP_1 *and* BP_2*, and add an output interface at the newly introduced end event and an input interface at the newly introduced start event:*

$I_{res} = I_1 \cup I_1 \setminus \{in(BP1), out(BP1), in(BP2), out(BP2)\} \cup \{start_{new}, end_{new}\}$ *where* $start_{new} \in \mathcal{E}_s, end_{new} \in \mathcal{E}_e$

These three operators are illustrated in Figure 2.

The semantics of these operators is defined in terms of token passing as follows: a token is generated by the start event and, through the outgoing sequence flow, reaches the different split gateways, that control the diverging of sequence flow. For the parallel operator, two parallel flow are generated, one for each outgoing arc, and a token produced on each output flow of the gateway. The tokens traverse in parallel the two branches and are synchronized by the merge parallel gateway. For the exclusive choice operator, a token is sent only on one of the output paths, based on the decision taken, activating just one of the two flows. The active path is then traversed by the token. The merge exclusive gateway must wait until the token from the active path arrives, and only then the sequence flow continues. For the choice operator, when the inclusive gateway is reached, for each outgoing sequence flow with a true condition, a token is generated and traverses that path. The merge inclusive gateway allows the process to continue only when tokens arrive from all incoming sequence flows where a token was generated before. After passing the merge gateways, the token is consumed by the end event, for each operator. The traces obtained, for each of these operators, are the same as for the Petri net operators.

3.5 Refinement Composition Operator

To apply this operator, two conditions need to hold on the input BPMN models: $BPMN_1$ must have an input or output composition interface at an internal task of the process; $BPMN_2$ must have an input interface at one of its end events and an output interface at the start event.

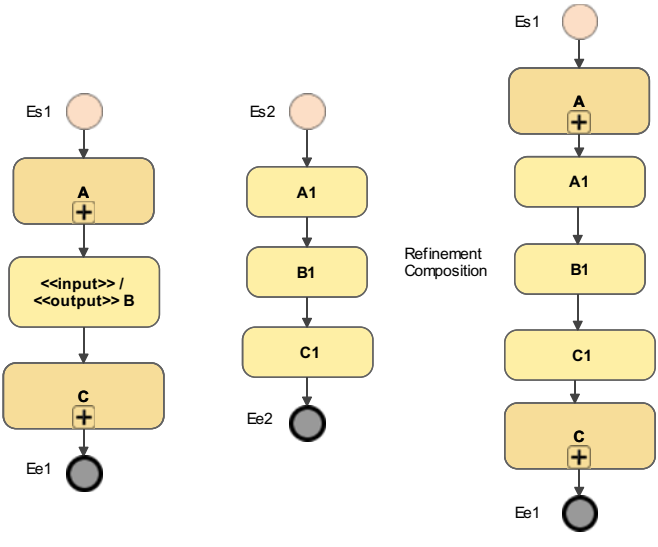


Fig. 3. Refinement operator for BPMN diagrams

We use of the following notations: let *comp1* be the task of $BPMN_1$ tagged with input or output composition interface

Definition 7. *Let* $BP_1 = (F_1, S_1, Ar_1, SF_1, MF_1, AS_1, I_1)$ *and* $BP_2 = (F_2, S_2, Ar_2, SF_2, MF_2, AS_2, I_2)$ *be two business processes. The result of applying the refinement composition operator on processes* BP_1 *and* BP_2*, denoted* $ref(BP_1, BP_2)$*, is a new BPMN process* $BP_{res} = (F_{res}, S_{res}, Ar_{res}, SF_{res}, MF_{res}, AS_{res}, I_{res})$ *where:*

The flow objects of the result contain the union of the flow objects of the input models, from which we remove the activity from BP_1 *tagged with a composition interface and the start end tagged end event from* BP_2*:*

$F_{res} = F_1 \cup F_2 \setminus \{comp1, in(BP2), out(BP2)\}$

The sequence flow of the result is obtained by first disconnecting the tagged activity from BP_1 *and then connecting the resulting two process fragments to process* BP_2 *from which we removed the start and tagged end event:*

$SF_{res} = SF_1 \cup SF_2 \setminus \{(pred(comp1), comp1), (comp1, succ(comp1)),$
$(in(BP2), succ(in(BP2))), (pred(out(BP2)), out(BP2))\} \cup$
$\{(pred(comp1), succ(in(BP2))), (pred(out(BP2)), succ(comp1))\}$

The swimlanes, artifacts, message flow and associations of the result are the union of their counterparts from the input processes:

$S_{res} = S_1 \cup S_2$, $Ar_{res} = Ar_1 \cup Ar_2$, $MF_{res} = MF_1 \cup MF_2$, $AS_{res} = AS_1 \cup AS_2$

The composition interface of the result contains the union of interfaces of the input models, from which we remove the tagged activity of BP_1*:*

$I_{res} = I_1 \cup I_2 \setminus \{comp1\}$

This operator is exemplified in Figure 3. We would like to add that a similar result can be obtained is using sub-processes in the following manner: in BP_1, activity B becomes a sub-process. This sub-process can then be expanded to contain process BP_1.

In terms of token passing, the semantics of this operator is the following: a new token is generated by the start event. Through the output sequence flow, it sequentially enables all the flow elements of BP_1, until it reaches the flow element tagged with a composition interface. Then, the token passes sequentially through all the flow elements of BP_2. After the last flow element of BP_2, the token moves to the flow element of BP_1 situated right after the tagged element. It then enables sequentially the rest of the flow elements of BP_1 until reaching end event of BP_1, where it is consumed. The same traces are obtained when the equivalent Petri net operator is applied.

3.6 Synchronization Composition Operator

To apply this composition operator, several conditions need to hold on the input processes:

- $BPMN_1$ and $BPMN_2$ have synchronization sets $Sync_1$ and $Sync_2$
- elements in $Sync_1$ must be tagged with input composition interfaces and those in $Sync_2$ with output composition interfaces

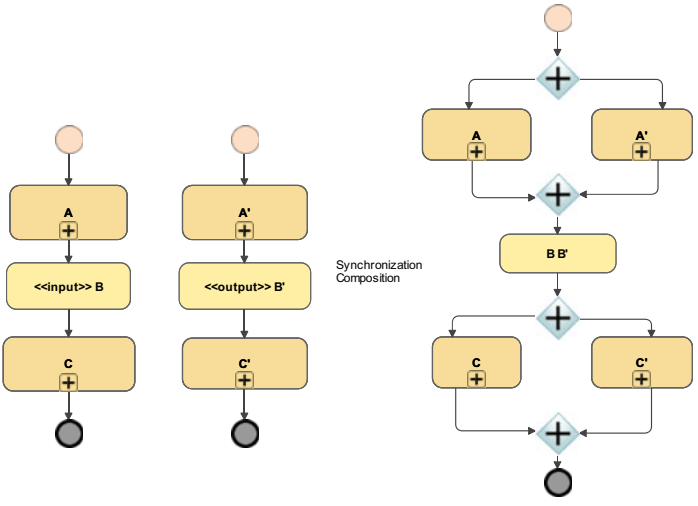


Fig. 4. Synchronization operator applied to 2 BPMN diagrams

– define the overall synchronization set (can only contain activities): $SyncSet = \{(x, y) | x \in Sync_1, y \in Sync_2\}$, with $x \in A_1, y \in A_2, A_1 \subseteq F_1, A_2 \subseteq F_2$

We use the following notation:
let $(act1_i, act2_j) \in SyncSet, act_{ij} \in \mathcal{A}, i, j = 1..|SyncSet|$

Definition 8. *Let* $BP_1 = (F_1, S_1, Ar_1, SF_1, MF_1, AS_1, I_1)$ *and* $BPMN_2 = (F_2, S_2, Ar_2, SF_2, MF_2, AS_2, I_2)$ *be two business processes. The result of applying the synchronization composition operator on processes* BP_1 *and* BP_2*, denoted* $synch(BP_1, BP_2)$*, is a new BPMN process* $BP_{res} = (F_{res}, S_{res}, Ar_{res}, SF_{res}, MF_{res}, AS_{res}, I_{res})$ *where:*

Activities of the result are the union of those of the input models, removing activities from SyncSet and adding new ones representing their merging:

$A_{res} = A_1 \cup A_2 \setminus \{act_i, act_j\} \cup \{act_{ij}\}$, act_{ij} *is the merging of* act_i *and* act_j

The events of the result are the union of events of the input models, from which we remove the start and end events and add new start and end events:

$E_{res} = E_1 \cup E_2 \setminus \{in(BP1), out(BP1), in(BP2), out(BP2)\} \cup \{start_{new}, end_{new}\}$, *where* $E_{res} \subseteq F_{res}, E_1 \subseteq F_1, E_2 \subseteq F_2, start_{new} \in \mathcal{E}_s, end_{new} \in \mathcal{E}_e$

To obtain the gateways of the result, we take the union of gateways of the input models and add new parallel gateways, depending on the number of elements in the synchronization set:

$G_{res} = G_1 \cup G_2 \cup \{g_i | g_i \in \mathcal{G}_p, i = 1..2 * |SyncSet|\}$

When the SyncSet has only one element, it defines two fragments (above, bellow) on each input process. The above fragments are put in parallel using parallel gateways; the same applies for the below fragments. The results thus obtained are then put in sequence, adding between them a new element that is the merging of synchronization elements:

$SF_{res} = SF_1 \cup SF_2 \setminus \{(in(BP1), succ(in(BP1))), (pred(out(BP1)), out(BP1)), (in(BP2), succ(in(BP2))), (pred(out(BP2)), out(BP2)), (pred(act1_1), act1_1), (act1_1, succ(act1_1)), (pred(act2_1), act2_1), (act2_1, succ(act2_1))\} \cup \{(start_{new}, g_1), (g_1, succ(in(BP1))), (g_1, succ(in(BP2))), (pred(act1_1), g_2), (pred(act2_1), g_2), (g_2, act_{12}), (act_{12}, g_3), (g_3, succ(act1_1)), (g_3, succ(act2_1)), (pred(out(BP1)), g_4), (pred(out(BP2)), g_4), (g_4, end_{new})$, *where* $SyncSet = \{(act1_1, act2_1)\}$

If $|SyncSet| > 1$, SF_{res} *we follow the same procedure, having also to compose the process fragments created between successive synchronization elements.*

The swimlanes, artifacts, message flow and associations of the result are the union of their counterparts from the input processes:

$S_{res} = S_1 \cup S_2$, $Ar_{res} = Ar_1 \cup Ar_2$, $MF_{res} = MF_1 \cup MF_2$, $AS_{res} = AS_1 \cup AS_2$

The composition interface of the result is constructed from the ones of the input models, from which we remove the elements from SyncSet.

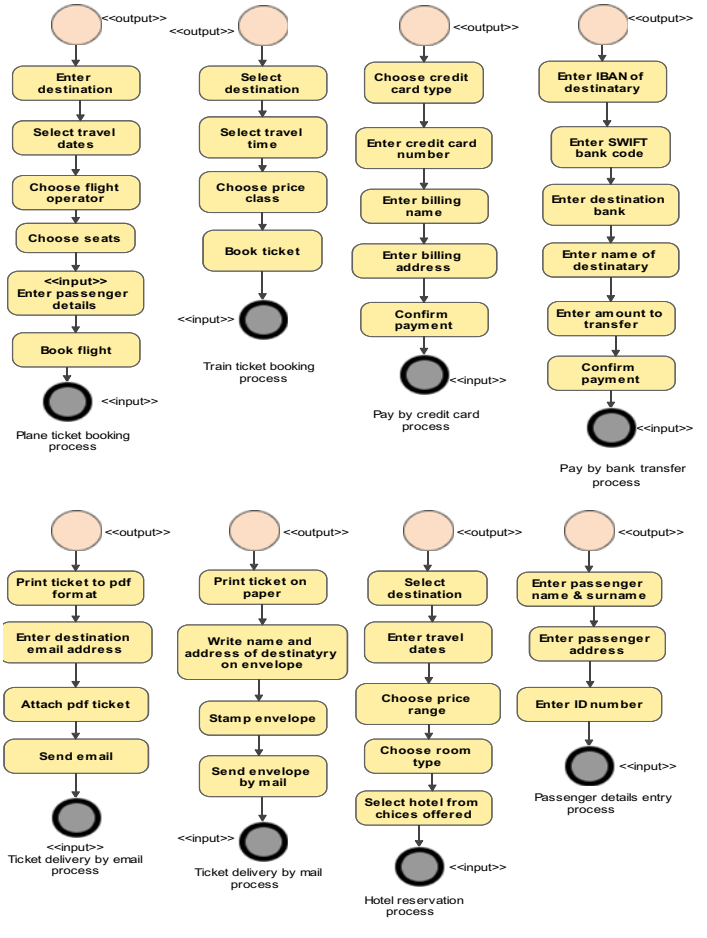


Fig. 5. Initial business processes used for composition

$$I_{res} = I_1 \cup I_2 \setminus \{act1_i, act2_j\}, i, j = 1..|SyncSet|, (act1_i, act2_j) \in SyncSet$$

A simple example describing how this operator is applied is depicted in Figure 4. We can observe that the refinement operator can be replaced by the combined application of other, less complicated operators: sequential and parallel.

In terms of token passing, the semantics of this operator is the following: a token is generated by the start event and through the outgoing sequence flow reaches the first split parallel gateway. Here, two tokens are generated for each outgoing sequence flow, which are executed in parallel. The flows synchronize at the merge parallel gateway. From its output sequence flow, the token is passed to the first merged synchronization element. Further, the same idea is applied for the process areas situated between and bellow synchronization elements, as they are put in parallel and tokens traverse them. After the last merge parallel gateway, the token is consumed by the end event.

4 Exemplification of Composition Operators

To better understand the composition operators introduced in Section 3, we explain how they can help construct complex business processes with the help of an example. In this example, we create an entire vacation reservation process using several composition operators. The process we want to create should cover the following aspects: explain how to book a plain or train ticket to the destination; describe the process of booking the hotel accommodation; describe the payment process, which can be performed either using a credit card or by bank transfer; finally, we want to receive the tickets bought either by email or by post. The user does not have to deal with all the complexity of creating such a process, as it is delegated to the composition operators.

To create the above mentioned process, we follow a separation of concerns approach and start by modelling a set of smaller business processes representing the individual activities to be performed: plane booking, train booking, hotel reservation, credit card payment, bank transfer, ticket delivery by email, ticket delivery by post, passenger data entry. All these business process fragments are depicted in Figure 5.

To obtain the desired business process, we need to apply several composition operators in successive steps. In a first composition step, the following intermediate business processes are obtained:

- *transportation booking*: we want to obtain a process that allows booking either a place ticket or a train ticket. Therefore, we apply the exclusive choice operator on the train booking and plane booking input processes.
- *payment*: we want to obtain a process allowing to choose either a credit card payment or a payment by bank transfer. We thus apply the exclusive choice operator on the credit card payment and bank transfer input processes.
- *ticket delivery*: we want a process that allows to deliver a ticket by email or by regular post or having both options. We apply the choice operator on the ticket deliver by email and ticket deliver by post input processes.

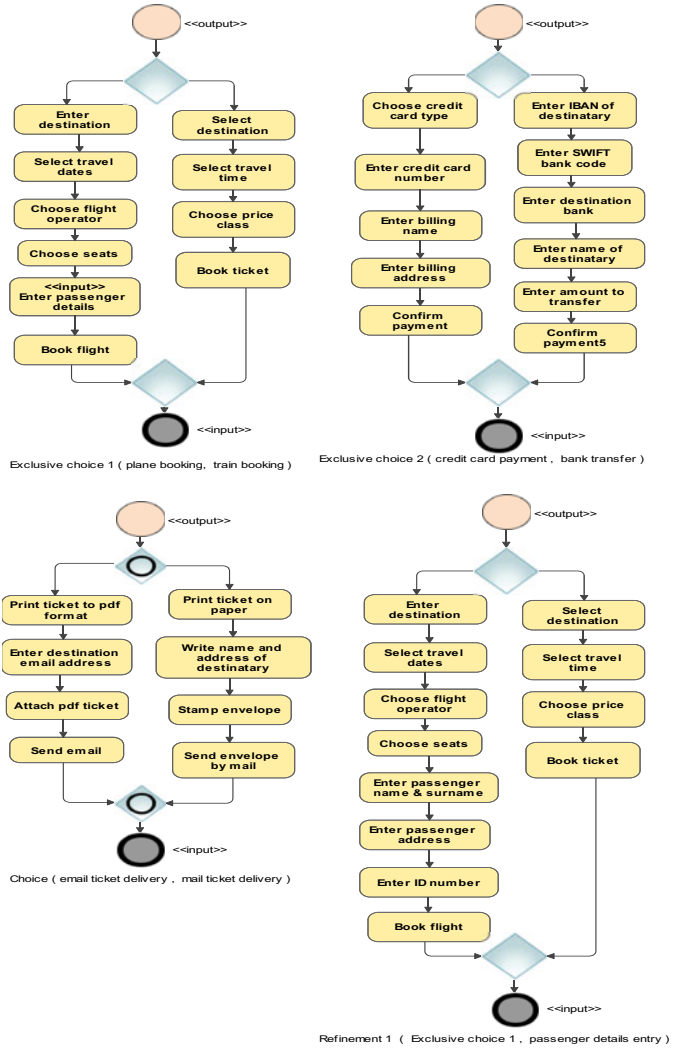


Fig. 6. First step of applying composition operators

– *detailed transportation booking*: we make use of the transportation booking process previously obtained. For thsi process, we require model details regarding the passenger data entry. Therefore, we apply the refinement operator on the transportation booking and passenger detail entry processes.

All the intermediate processes obtained are described in Figure 6.

In a second composition step, we make use of the intermediate business processes previously obtained and construct a new set of intermediate processes that bring us closer to the end result. The newly created processes are:

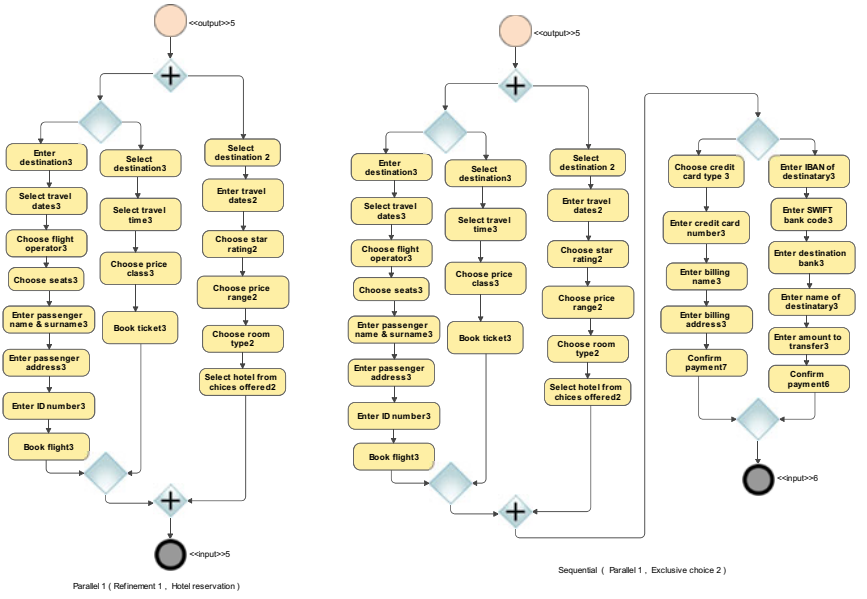


Fig. 7. Second step of applying composition operators

- *transportation and hotel booking*: we want a process that allows to book a hotel accommodation but in the same time to reserve a train or plane ticket to the desired destination. This process is obtained by applying the parallel operator on the transportation booking and hotel booking input processes.
- *transportation, hotel booking and payment*: we make use of the previous process, but in addition we now also require the possibility to pay for the reservations. Therefore, we use the sequential composition operator to compose the transportation and hotel booking process with the payment process.

These intermediate processes obtained during the second composition step are depicted in Figure 7.

To obtained the final result modelling the entire vacation reservation process, we need to compose, using the sequential operator, the following two intermediate business processes: *transportation, hotel and payment* process with the *ticket delivery* process.

5 Conclusions

Modern enterprises are constantly changing and evolving. Implicitly, business processes have to accommodate such changes. Therefore, business process models are becoming exponentially more complex. This complexity makes them hard to understand and use, and renders their maintenance and evolution more difficult and at higher costs.

Business process composition is a viable solution to the above-mentioned issues. At the heart of every process composition are composition operators. Most languages adopt and provide a fixed set of composition operators.

BPMN, the standard language for modelling business processes, is lacking such composition operators. Throughout this paper, we proposed to extend the BPMN standard with a well defined set of composition operators. They are formally defined and their semantics is briefly presented in terms of token passing. In addition, we extend BPMN with composition interfaces, to explicitly mark the places where a process can be composed with other ones. Composition interfaces are also used when applying the proposed operators.

We already started to define other composition operators for BPMN: unordered sequence, parallel composition with communication, discriminator composition or fragment insertion. We are also interested in determining which properties of a business process are conserved by the different operators. As another possible research direction, we want to study how constructing a business process using a compositional approach can help its verification and validation.

References

1. van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M.: Business Process Management: A Survey. In: van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M. (eds.) BPM 2003. LNCS, vol. 2678, pp. 1–12. Springer, Heidelberg (2003)
2. Allweyer, T.: BPMN 2.0. BoD (2010)
3. Anisimov, N.A., Golenkov, E.A., Kharitonov, D.I.: Compositional petri net approach to the development of concurrent and distributed systems. Program. Comput. Softw. 27, 309–319 (2001)
4. Best, E., Fraczak, W., Hopkins, R., Klaudel, H., Pelz, E.: M-nets: an algebra of high-level petri nets with an application to the semantics of concurrent programming languages. Acta Informatica 35, 813–857 (1998)
5. Best, E., Lavrov, A.: Generalised composition operations for high-level petri nets. Fundam. Inf. 40(40), 125–163 (1999)
6. Davenport, T., Short, J.: He new industrial engineering: Information technology and business process redesign. Sloan Management Review 31, 11–27 (1990)
7. Davis, R.: Process reuse do we understand what it really means? (February 2010), http://www.bptrends.com/publicationfiles/SIX
8. Devillers, R., Klaudel, H., Pelz, E.: An algebraic box calculus. Journal of Automata, Languages and Combinatorics 5 (2000)
9. Eshuis, R., Norta, A.: Business process composition. In: Mehandjiev, N., Grefen, P. (eds.) Dynamic Business Process Formation for Instant Virtual Enterprises. Advanced Information and Knowledge Processing, pp. 93–111. Springer, London (2010)
10. Esparza, J., Silva, M.: Compositional Synthesis of Live and Bounded Free Choice Petri Nets. In: Groote, J.F., Baeten, J.C.M. (eds.) CONCUR 1991. LNCS, vol. 527, pp. 172–187. Springer, Heidelberg (1991)
11. Hashemian, S., Mavaddat, F.: Composition algebra: Process composition using algebraic rules. In: Proceedings of the Third International Workshop on Formal Aspects of Component Software (September 2006)
12. Havey, M.: Essential Business Process Modeling. O'Reilly Media, Inc. (2005)

13. Heineman, G.T., Councill, W.T.: Component-based software engineering: putting the pieces together. Addison-Wesley Longman Publishing Co., Inc. (2001)
14. Istoan, P.: Consistency rules for bpmn processes (2011), http://wiki.lassy.uni.lu/@api/deki/files/490/=Consistency.pdf
15. Jensen, K., Kristensen, L.M., Wells, L.: Coloured petri nets and cpn tools for modelling and validation of concurrent systems. Int. J. Softw. Tools Technol. Transf. 9, 213–254 (2007)
16. Markovic, I., Pereira, A.C.: Towards a Formal Framework for Reuse in Business Process Modeling. In: ter Hofstede, A.H.M., Benatallah, B., Paik, H.-Y. (eds.) BPM Workshops 2007. LNCS, vol. 4928, pp. 484–495. Springer, Heidelberg (2008)
17. Moreira, A., Rashid, A., Araujo, J.: Multi-dimensional separation of concerns in requirements engineering. In: Proceedings of 13th IEEE International Conference on RE, pp. 285–296. IEEE Computer Society, Washington, DC (2005)
18. Mosser, S., Blay-Fornarino, M., France, R.: Workflow Design using Fragment Composition - Crisis Management System Design Through ADORE. In: Transactions on AOSD VII. LNCS, vol. 6210, pp. 200–233. Springer, Heidelberg (2010)
19. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4), 541–580 (1989)
20. OMG: Business process model and notation (bpmn) version 2.0 (January 2011), http://www.omg.org/spec/BPMN/2.0/PDF/
21. Pankratius, V., Stucky, W.: A formal foundation for workflow composition, workflow view definition, and workflow normalization based on petri nets. In: Proceedings of the 2nd Asia-Pacific Conference on Conceptual Modelling, vol. 43, pp. 79–88. Australian Computer Society, Inc. (2005)
22. Petri, C.A.: Kommunikation mit Automaten. Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, Bonn (1962)
23. Straw, G., Georg, G., Song, E.J., Ghosh, S., France, R.B., Bieman, J.M.: Model Composition Directives. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) UML 2004. LNCS, vol. 3273, pp. 84–97. Springer, Heidelberg (2004)
24. White, S.A.: Workflow patterns with bpmn and uml. IBM (January 2004)
25. Zhovtobryukh, D.: A petri net-based approach for automated goal-driven web service composition. Simulation 83, 33–63 (2007)

Relaxing B Sharing Restrictions within CSP||B*

Arnaud Lanoix[1], Olga Kouchnarenko[2], Samuel Colin[3], and Vincent Poirriez[4]

[1] LINA CNRS and Nantes University, Nantes, France
arnaud.lanoix@univ-nantes.fr
[2] FEMTO-ST CNRS and University of Franche-Comté, Besançon, France
olga.kouchnarenko@univ-fcomte.fr
[3] SafeRiver, France
scolin@hivernal.org
[4] LAMIH CNRS and University Lille Nord de France, Valenciennes, France
vincent.poirriez@univ-valenciennes.fr

Abstract. This paper addresses the issue of state sharing in CSP||B specifications: B machines controlled by various CSP parts are supposed not to refer to, share or modify the same state space. However, some kinds of B state sharing can be allowed without creating inconsistencies in CSP||B specifications. To achieve this, we present a B-based solution for allowing architectures with B state sharing in the CSP||B components. We show that the inconsistencies in state sharing can be identified by translating the CSP controllers into B specifications and then using a more refined consistency checking process. We also hint at possible extensions towards other CSP||B architectural patterns with various types of sub-components sharing.

Keywords: CSP||B, sharing, architecture, consistency, rely-guarantee.

1 Introduction

In this work we address the question of how to safely reuse already-developed B component models in which there is a common and shared part when developing a CSP||B model. The problem of sharing is known to be difficult in the framework of the B method whereas it is naturally supported by the CSP formalism.

The present work is motivated by an example which arose during the process of assembling already formally specified and proved components. In the context of the TACOS project, we modelled a multi-agent system of a convoy [1] while a complex B model of a location component was also independently designed [2]. Integrating the latter into the former appears to be problematic because the resulting assembly risks breaking the consistency of the whole vehicle component, as state sharing is involved. Machine sharing, like in the location component, is not valid at the CSP||B level. In fact, such an architecture goes against

* Work supported by the ANR-06-SETI-017 project: "TACOS: Trustworthy Assembling of Components: frOm requirements to Specification" (http://tacos.loria.fr).

T. Gschwind et al. (Eds.): SC 2012, LNCS 7306, pp. 35–50, 2012.

the general (and well-known) "one controller≡one machine" CSP||B constraint. Moreover, although more recent results allow several controllers to share a B machine, like in [3], they do not permit to deal with our case study.

Several relevant architectures involve B state sharing which can happen because of sharing a B machine by other B machines. This is the reason why we focused primarily on using notions coming from the B setting such as its modularity links. In a nutshell, our approach is about characterising the links between controllers and machines as *seeing* or *importing* links in the B sense. It then becomes possible to consider the whole CSP part of the system as a single B machine and to use the B constraints upon this "transformed" system to decide whether the shared B machines of the system can have their invariants broken or not.

Unlike [3,4], the novelty of our approach is thus bringing B sharing *to the B level*. Indeed, in [4], Evans et.al used CSP controllers "augmented" with a B part to perform automatic consistency and non-discrimination checks of CSP||B models. In that work, determining which parts of a CSP||B analysis can be handled within the B method has been left aside as a future work direction. The approach advocated in the present paper deals mostly with the B part, hence it can be viewed as complementary. Those works and ours could thus be used together to bring state sharing at every level of the CSP||B formalism. More precisely, we show how to use the B modularity constraints to allow CSP||B models with multiple controllers for a B machine or with a single controller for multiple, and possibly shared, B machines. We then propose a refined consistency checking of CSP||B based on such architectural patterns.

Layout of the paper. Before introducing a platoon example and a part of its modelling in Section 3, we present the necessary formalisms, concepts and tools in Sect. 2. Our main contributions are in Sect. 4 and 5. We propose 1) a method—based on the B modularity—for detecting inconsistent CSP||B architectures, and 2) a refinement of CSP||B consistency check requirements based on architectural patterns. In addition, we propose extensions for addressing the verification of more complex cases. Finally, conclusions and assessments are drawn in Sect. 6, combined with related work on state sharing in CSP||B and B.

2 Concepts and Tools for CSP||B Components

The B machines specifying components are open modules which interact by the authorised operation invocations. CSP describes processes, i.e. objects or entities which exist independently, but may communicate with each other. When combining CSP and B to develop distributed and concurrent systems, CSP is used to describe execution orders for invoking the B machines operations and communications between the CSP processes.

2.1 B Machines

B is a formal software development method used to model and reason about systems [5]. The B method has proved its strength in industry with the development

of complex real-life applications such as the Roissy VAL [6]. The principle behind building a B model is the expression of system properties which are always true after each evolution step of the model, the evolution being specified by the B operations. The verification of a model correctness is thus akin to verifying the preservation of these properties, no matter which step of evolution the system takes.

The B method is based on first-order logic, set theory and relations. A strength of the B method is its stepwise refinement feature: each refinement makes a model more deterministic and also more precise by introducing programming language-like features, until the code of the operations can actually be implemented in a programming language.

Let us assume here that the initialisation is a special kind of operation. In this setting, a B architecture is *consistent* if the following conditions hold [5,7]:

- Each machine has its invariant preserved by its operations, i.e. the model is *consistent*.
- Each refinement or implementation can replace the B machine it refines.

Both items above are semi-local: the proof obligations correspond to a local reasoning, but the machines can use operations of included or seen machines. It must then be verified that these operations are correctly used: this is done implicitly when operation invocations are expanded into their respective bodies. This ensures that the proof obligation contains a sub-goal for checking that the invoked operation is indeed called within its precondition.

Support tools such as B4free (`http://www.b4free.com`) or AtelierB (`http://www.atelierb.eu`) automatically generate Proof Obligations (POs) to ensure the consistency [5]. Some of them are "obvious" POs which are automatically discharged whereas the normal POs have to be proved interactively if it was not done fully automatically.

Modularity in B. The B project architecture can be handled through some specific clauses **SEES**, **INCLUDES** and **USES** that allow a machine to list its seen machines, included machines or used machines, respectively. The **IMPORTS** clause corresponds to **INCLUDES** for an implementation model. A B architecture must respect some *modularity* constraints. For instance, one machine cannot end up being included or imported twice by two different inclusion paths, as this could break the invariant. In [8] the modularity constraints in [5] have been proved to be not strong enough, because intermediate **SEES** links could hide the fact that a machine could be modified through refinement. In [7], a modularity constraint to ensure no invariant breakage and no interference by a machine with a seen machine through another indirect path, is given:

Theorem 1. $(uses; can\_alter) \cap ((imports; s^+) \cup (sees; s^*)) = \emptyset$

with *sees* being the set of couples (M_1, M_2) where the implementation of M_1 "sees" the machine M_2, *imports* a similar set where the implementation of M_1 "imports" M_2, s the set where M_1 directly "sees" M_2, $uses = sees \cup imports$

and $can\_alter = (uses^*; imports)$. The ; operator corresponds to the B relation composition, * to the B reflexive transitive closure, and $^+$ to the transitive closure. No double importation and no violation of the constraint of Theo. 1 ensure no invariant breakage and no interference by a machine with a seen machine through another indirect path.

When taking into account all implicit hypotheses about B modularity [7], the formula can actually be simplified into the following shape: $can\_alter \cap sees = \emptyset$. We pointed out this modularity constraint because of the role it plays in our contribution in Sect. 4.

2.2 Communicating Sequential Processes (CSP)

CSP allows the description of entities, called processes, which exist independently but may communicate with each other. Thanks to dedicated operators it is possible to describe a set of processes as a single process, making CSP an ideal formalism for building a hierarchical composition of components. CSP is supported by the FDR2 model checker (`http://www.fsel.com`). This tool is based on the generation of all the possible states of a model and the verification of these states against a desired property.

The denotational semantics of CSP is based on the observation of process behaviours. Three kinds of behaviours [9] are observed and well suited to express the properties:

- traces, i.e. finite sequences of events, for safety properties;
- stable failures, i.e. traces augmented with a set of unperformable events at the end thereof, for liveness properties and deadlock-freedom;
- failures/divergences, i.e. stable failures augmented with traces ending in an infinite loop of internal events, for livelock-freedom.

Each kind of behaviours gives rise to a notion of process refinement defining a particular semantical framework [9].

2.3 CSP||B Components

In this section, we sum up the works by Schneider and Treharne on CSP||B. The reader interested in theoretical results is referred to [3,10] and the abundant CSP||B literature referenced therein; for case studies, see for example [11,12].

Specifying CSP Controllers. In CSP||B architecture (as depicted Fig. 1), the B part is specified as a B machine without any restriction, while the controller is a CSP process, called a CSP controller, defined by the following subset of the CSP grammar:

$$\begin{array}{l} \mathsf{P} ::= \mathsf{c}\ ?\ \mathsf{x}\ !\ \mathsf{v}\ \rightarrow \mathsf{P} \mid \mathsf{ope}\ !\ \mathsf{v}\ ?\ \mathsf{x} \rightarrow \mathsf{P} \\ \quad \mid \mathsf{b}\ \&\ \mathsf{P} \mid \mathsf{P} \,\Box\, \mathsf{P} \mid \textbf{if}\ \mathsf{b}\ \textbf{then}\ \mathsf{P}\ \textbf{else}\ \mathsf{P} \mid \mathsf{S}(\mathsf{p}) \end{array}$$

The process $\mathsf{c}\ ?\ \mathsf{x}\ !\ \mathsf{v} \rightarrow \mathsf{P}$ can accept input x and output v along a communication channel c. Having accepted x, it behaves as P.

Machine channels are introduced in CSP controllers to provide the means for controllers to synchronise with the B machine: for each B operation $x \leftarrow ope(v)$, there can be a channel ope ! v ? x in the controller corresponding to the operation call: the output value v from the CSP description corresponds to the input parameter of the B operation, and the input value x corresponds to the output of the operation. A controlled B machine can only communicate on the machine channels of its controller.

Remark 1. CSP‖B components must respect the "one controller≡one machine" constraint (as shown in Fig. 1): controlled B machines are not allowed to share states, i.e. they cannot *see* or *import* the same machines. Then, the CSP‖B model necessarily respects the B modularity constraints (Theo 1, Sect. 2.1).

The behaviour of a guarded process b & P depends on the evaluation of the boolean condition b: if it is true, it behaves as P, otherwise it is unable to perform any events. In some works (e.g. [10]), the notion of *blocking assertion* is defined by using a guarded process on the inputs of a channel to restrict these inputs: c ? x & E(x) → P.

The external choice P1 □ P2 is initially prepared to behave either as P1 or as P2, with the choice made on the occurrence of the first event. The conditional choice **if** b **then** P1 **else** P2 behaves as P1 or P2 depending on b. Finally, S(p) expresses a recursive call. Finally, in addition to the expression of simple processes, CSP provides parallel composition operators to combine them.

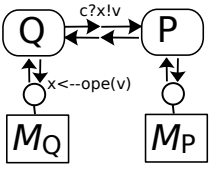


Fig. 1. CSP‖B components

Verifying CSP‖B Components. The main problem with combined specifications is their *consistency*: CSP and B parts should not be contradictory. Let us assume a CSP‖B compound $(P\|M_P)$. The verification process to ensure the consistency of $(P\|M_P)$ consists in verifying the following conditions [10]:

1. Check the *consistency* of M_P with B4Free or Atelier-B for instance,
2. Check the *deadlock-freedom* (in the stable-failures model) and *divergence-freedom* of P with FDR2,
3. Check the *divergence-freedom* of $(P\|M_P)$ (see below),
4. By way of [10, Theorem 5.9] and the fact that P is deadlock-free, the *deadlock-freedom* of $(P\|M_P)$ in the stable failures model is deduced.

The given results are also generalised in [10] to a collection of B machine-CSP process couples. The whole CSP‖B architecture must also respect the sharing constraint recalled Remark 1.

Ensuring the Divergence-Freedom of CSP‖B Components. Originally, the technique for ensuring the *divergence-freedom* of a controlled machine $(P\|M_P)$ involved the stating of a Control Loop Invariant (CLI) and its verification [13,14]. Fortunately, the above technique has evolved into a more general

and less cumbersome one. Evans & Treharne [11] have defined a fixed-point rule for deducing the non-divergence of a controlled machine $(P \| M_P)$.

To sum up, the fixed-point rule procedure is based on the satisfaction by the controller P of a uniform property $\overrightarrow{every}(p)(S)(T)$, where p is an event predicate and S, T are states (e.g. predicates expressed in the B set theory): P **sat** $\overrightarrow{every}(p)(S)(T)$. See [11,3] for more details, with a PVS implementation.

That fixed-point rule relates the use of a CLI for verifying the divergence-freedom of a controller to uniform properties for CSP controllers. The use of uniform properties for CSP controllers lifts the need for preprocessing as done earlier with the explicit construction of a CLI, and it generalises the parallel composition of CSP controllers.

In [3], the authors deduced the divergence-freedom of $P \| Q$ by verifying the *non-interference*, i.e. a property which expresses that P does not interfere with the traces of Q, denoted as $non\_interference(p, P, Q)$. Then, they deduced:

Property 1.

$$\text{If } \left\{ \begin{array}{l} non\_interference(p, P, Q) \\ \wedge\ non\_interference(p, Q, P) \\ \wedge\ P \textbf{ sat } \overrightarrow{every}(p)(S)(T) \\ \wedge\ Q \textbf{ sat } \overrightarrow{every}(p)(S)(T) \end{array} \right\} \text{ then } P \| Q \textbf{ sat } \overrightarrow{every}(p)(S)(T)$$

3 Motivating Case Study

This section presents an example which arose during the process of assembling already formally specified and proved components. In [1] a convoy, the so-called platoon, of autonomous vehicles (depicted in Fig. 2) was fully specified and validated in the framework of the CSP||B methodology. The behaviour of this system is described *in extenso* in [15]. In the context of this paper we are more concerned with the part of the model limited to a single vehicle. Figure 3 illustrates a single vehicle, one element of the platoon. Its formal study can be found in [1].

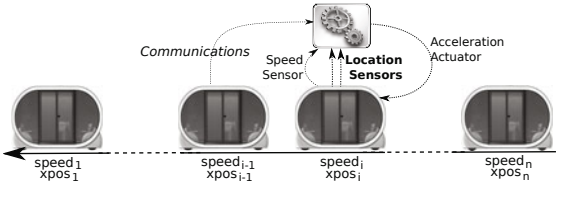


Fig. 2. A platoon of autonomous vehicles as a multi-agent system

In figures the conventions are as follows: the rounded boxes depict CSP controllers, whereas the others show B machines, with the plain arrows between CSP processes or between a CSP controller and a B machine being read-write links, and dotted arrows being read-only links.

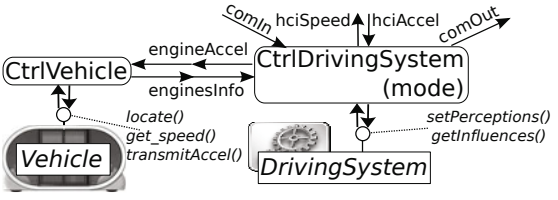


Fig. 3. Abstract CSP||B vehicle

This first CSP||B specification was refined in [16]. The resulting more detailed specification was proved to refine – in the traces/failures model of CSP – the previous specification. In [16] the refined specification involves several controllers (instead of the only CtrlVehicle controller) equipped with B machines. It also contains an abstract model of a location component answering the locate() B method by determining the geographic position of the physical vehicle.

In the framework of the TACOS project, more concrete B specifications of the location component have been independently proposed in [2]: an enhanced realistic pure B model of the vehicle (with focus on the location problem) was derived from the requirements specified using the KAOS method [17].

One of the introduced safety requirements is that location sensors would be an assembly of several so-called *raw positioning components* based on different technologies (GPS, Wifi, GSM, Visual sensors,...). Each raw positioning sensor provides a chronologically ordered set of locations. The sets of all components must be merged. In addition, to (in)validate the provided data, an actual speed and acceleration can be used. It allows keeping only the possible, i.e. consistent, locations, and removing the inconsistent ones.

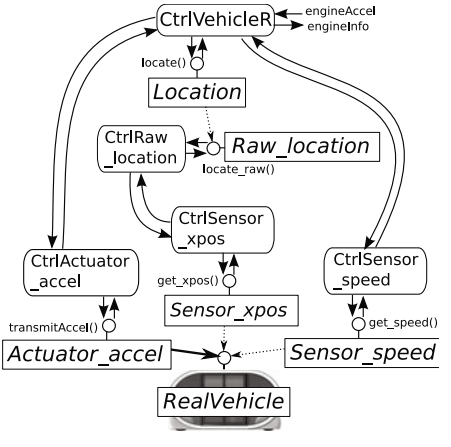


Fig. 4. Enhanced CSP||B vehicle

Figure 4 displays a simplified CSP||B vehicle model enhanced with the Location component[1]. In this model, Actuator_accel and Sensor_speed are separate B machines. This is the result of differentiating acceleration values *as they are passed to the engine* and acceleration values *as they have been effectively applied by the engine*. We want to emphasise the fact that in Fig. 4, some of the CSP controllers share B machines. For example, CtrlVehicleR and CtrlRaw_location share a view on Raw_location. Consequently, the consistency of the whole CSP||B vehicle component risks to

[1] A detailed version of this paper with an appendix depicting a bigger and more complete version of the case study is available at http://tacos.loria.fr/drupal/?q=node/83.

be broken because of state sharing. The question we are interested in is: *"Is it possible to relax CSP||B restrictions on the architecture of the B part so that we can indeed realise the needed integration?"*

4 B-Based State Sharing within CSP||B

As recalled in Remark 1 (and in Fig. 1), a CSP||B architecture disallowed any sharing of B machines. This way, there is no risk for the invariant of the nonexistent shared machine to be broken, nor for any machine or controller to suffer from interferences from an adjacent controller-machine pair. However, Figure 5 shows several relevant architectures involving B state sharing. Machine sharing can happen because of sharing by other B machines as in (a), (b) and (d) or because of sharing by several controllers as in (c).

Our goal, as exhibited in Sect. 3, is to relax restrictions on the architecture of the B part of a CSP||B model. In this section, we show that it is possible to express the way the controlled B machines are used by the CSP part in terms of B modularity links, and to include them in the B modularity checking, to allow B state sharing in CSP||B.

More precisely, we are concerned with architectures (a) and (b), with some considerations about (d): the novelty of our approach is thus bringing B sharing *to the B level*. This is the reason why we focused primarily on using notions coming from the B setting such as its modularity links. In a nutshell, our approach is about characterising the links between controllers and machines as *seeing* or *importing* links in the B sense. It then becomes possible to consider the whole CSP part of the system as a single B machine and to use the B constraints upon this transformed system to decide whether the shared B machines of the system can have their invariants broken or not.

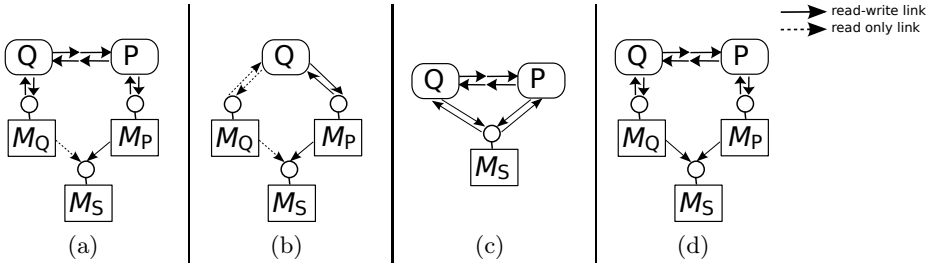


Fig. 5. Several architectures depicting the sharing of B machines

4.1 B Modular Characterisation of CSP Control

We want to characterise in B terms, the *machine channels*, i.e. the CSP-controlled operations. In [5] Abrial indicates that an operation can be callable, callable in inquiry or not callable. In the first case, such as for an importation link, the

called operation can modify the state of the imported machine. In the second case, it cannot: it is the case for a seen machine, whose such inquiry operations allow an external machine to observe the state of the seen machine. The third case corresponds to more specific modularity links, such as the **USES** link.

In modular B terms, the CSP control of a B machine can be viewed as a kind of **INCLUDES** or **IMPORTS** links: the operations triggered by the CSP part of the system can modify the variables of the controlled machines. A first guideline would thus be that we would consider CSP‖B "links" as **IMPORTS** links. We nonetheless can do a finer analysis: it may be the case that a CSP controller never modifies the state of its controlled machine but merely passes around the result of calculations, for instance. We could thus characterise CSP‖B links with the following definition:

Definition 1. *If all the operations of a B machine triggered by its CSP controller are inquiry operations in the B sense, then we say that the CSP controller* ***SEES*** *its controlled B machine. Otherwise, we will say that the CSP controller* ***IMPORTS*** *its controlled B machine.*

Detecting whether an operation is an inquiry operation is rather straightforward: it is defined as being an operation *not* changing the variables of its component [18, Annex E]. Finding if an operation is an inquiry operation can thus be done at the syntactic level, by detecting whether the variables of the machines appear in the left members of the modifying substitutions of the considered operation.

This way we can characterise the CSP controls of the B part in terms of the modularity of B. Then, we want to express the CSP part of a CSP‖B system as a B entity, to check the B modularity constraints on the whole CSP‖B system.

4.2 From CSP to B Modularity

It is well-known that a CSP system can be translated into B using results in [19,20]. We might stop here and use this translation, with adding what is needed for translating the CSP‖B links. Instead we go further by exploiting the fact that the verifications to correctly share a B machine are lifted to the architecture of the project. Indeed, these verifications can be done through two B-based steps:

- Verifying that the way the variables and operations are used matches the kind of modularity link that is used, for each machine. For instance, verifying that the operations of a seen machine are inquiry operations.
- Verifying that the architecture respects the modularity constraints imposed by the B method, such as the constraint in Sect. 4.1.

Because we characterised the CSP→B links by means of the **IMPORTS** or **SEES** links depending on what operations the controllers use, we obtain the first step by virtue of construction. We are left with the second step: the content of the B machine does not matter for this step. This means that the content of the CSP system translated into B does not matter either.

Property 2. Let the CSP part be represented by a single B machine, and the links between CSP controllers and B machines be characterised either as **IMPORTS** links or as **SEES** links. If the resulting system respects the modularity constraints of B, then no shared machine in the B part of the system can have its invariant broken.

Proof. (Sketch) (i) Let us assume that the translation from CSP into B is correct. It is based on the results in [20]. (ii) The interactions between CSP and B parts can be characterised in terms of the B modularity (see Sect. 4.1).
Consequently, if the whole system expressed in B thanks to (ii) satisfies the modularity constraints of B given by Theo 1, Sect. 2.1 then, by (i), the CSP‖B system also satisfies the modularity constraints, and no shared B machine has its invariant broken. Obviously, the last point only concerns the B part. □

This property is a direct consequence of lifting all the CSP parts of the system into a B setting: any B architecture that respects the modularity constraints ensures this property.

Thanks to our proposals, the process for checking that the B part of a CSP‖B system with sharing of B machines is consistent becomes as follows:

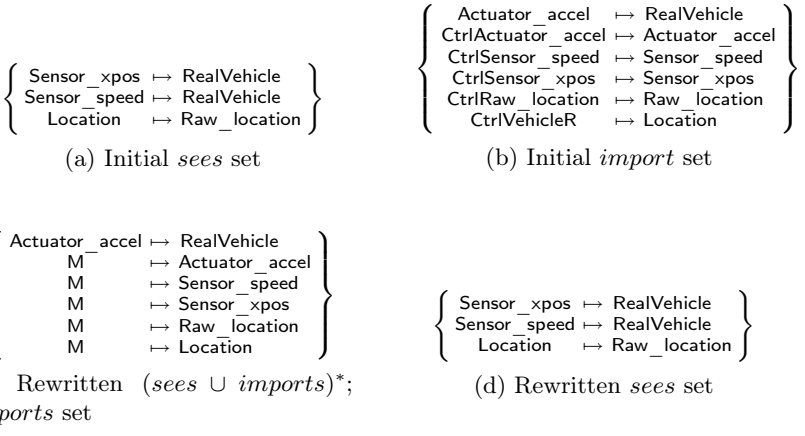
1. Characterise the links of each controller to its controlled machine in a B fashion (**IMPORTS** or **SEES**).
2. Represent the whole CSP system (with the CSP controllers) as a single B machine (using *csp2b* for instance [19,20]) which imports or sees the various controlled machines, depending on how the links have been characterised.
3. Check the resulting pure B architecture with usual B tools, B4free or Atelier-B for instance.

Notice that Property 2 is a sufficient but not necessary condition. If the tool checking is successful, then the way the B machine is shared in the whole CSP‖B system is consistent. If it fails, then the shared machines face a potential invariant breakage. The example in the next section illustrates this step.

4.3 Application to the Vehicle System

Let us consider again Fig. 4. Let M be the B entity corresponding to the CSP processes (or controllers): CtrlVehicleR, CtrlActuator_Accel, CtrlSensor_Speed, CtrlRaw_location and CtrlSensor_xpos. Although there is no direct link between CtrlVehicleR and CtrlRaw_location, they are still executed in parallel and could cause invariant breakage in a commonly shared B machine. Let us analyse this.

Let us write the *sees* and *imports* sets depicted by Figs 6a and 6b for calculating whether the architecture respects the B modularity constraints. We kept the names of the differentiated CSP controllers/processes with respect to Fig. 4 instead of using M. The controller→machine links are importation links because the machines are modified, as they are used for backing up the passed value in a log. Now after having rewritten the CSP controllers or processes into M, the final $(sees \cup imports)^*; imports$ set which contains the possibly, and indirectly,



(a) Initial *sees* set

(b) Initial *import* set

(c) Rewritten $(sees \cup imports)^*; imports$ set

(d) Rewritten *sees* set

Fig. 6. *sees*, *imports* and $(sees \cup imports)^*; imports$ sets

modified machines is given Fig. 6c. Note that M will never be a target, because the whole CSP part will always be a source of inclusion/sight towards B machines. The intersection of the relations in Figs 6d and 6c is empty, hence the architectural B criterion (Sect. 2.1 and 4.1) is satisfied.

The divergence-freedom of the controlled machines is also respected. Although the code of the machines is not shown here, it is very simple as we do not make strong assumptions about the passed values at the moment. The various preconditions of the machines are thus merely for typing the variables.

5 Ensuring Divergence-Freedom of Shared B Machines

Control loop invariant checking [14,10] or uniform property verification [3] ensure that a controlled B machine never diverges, i.e. its operations are never called outside their preconditions, through the triggering of its operations by the CSP controller.

Let us consider the architecture of Fig. 5(a): the M_S machine is imported by M_P and seen by M_Q, which are themselves imported by their respective controllers P and Q. This architecture is sound with respect to the architectural constraints of Sect. 4.1, hence M_S will not have its invariant broken.

Let us now imagine that an operation ope_q of M_Q references some variable of M_S in its precondition, e.g. in the shape of $x_S > 0$. The invariant of M_Q relies thus indirectly upon the strict positivity of x_S. Let us suppose that checking the consistency of $Q \| M_Q$ does not show any problem. Then, what happens if M_P, because it includes or imports M_S, triggers an operation that makes $x_s = 0$? Then the precondition of ope_q becomes invalid, even though consistency checking did not exhibit the problem. The problem depicted here is typically a problem of *non-interference*, and the consistency checking approach as presented in Sect. 2 is not sufficient.

Let us notice that in [3] the authors encountered a similar problem for related but different reasons. Their non-interference Property 1 recalled Sec. 2 is used in a case similar to the architectural case illustrated by Fig. 5(c) because the both controllers "import" the shared machine, hence can interfere with each another.

Fortunately, it turns out that Property 1 can be simplified in our architectural case depicted Fig. 5(a). Indeed, we know that the shared machine is effectively imported only by one controller, because of the B rule stating that a machine can only be imported once. Hence we know that this shared machine will be unaffected by all other controllers: they will only ultimately be allowed to refer to the shared machine through **SEES** links, hence they can never modify the shared machine. We thus integrate this specificity in Property 1, leading to:

Property 3. If P is a controller that ends up importing a shared machine (Fig. 5(a)), and

$$\left\{\begin{array}{l} non\_interference(p,P,Q) \\ \wedge\ P\ \mathbf{sat}\ \overrightarrow{every}(p)(S)(T) \\ \wedge\ Q\ \mathbf{sat}\ \overrightarrow{every}(p)(S)(T) \end{array}\right\} \text{ then } P\|Q\ \mathbf{sat}\ \overrightarrow{every}(p)(S)(T)$$

As the non-interference property is trivially verified for Q with P thanks to the knowledge about the architecture of the system, we simply removed it. The other non-interference properties must be kept: because P imports the shared machine, it can still have an effect on the other controllers that see the shared machine.

Proof. (Sketch) Let assume without loss of generality that the whole CSP‖B system satisfies the modularity constraints (Sect. 4.1). As a consequence, in our architectural case only P can write into M_S. Hence Q (or other seeing controllers) can only use non-modifying operations of M_S. As a result, Q does not interfere with the P behaviour. This can be shown (i) by induction on the traces tr—universally specified in $every(p)(S)(T)(tr)$—of invocations by P of operations of the controlled B machines M_P and M_S, and (ii) by analysis of the effect of M_S operations called by Q via M_Q: as operations are non-modifying there is no interference in this case. On the other hand, because of the $non\_interference(p,P,Q)$ hypothesis, P does not interfere with the Q behaviour, and we are done. □

Thanks to our proposals, the restriction on state sharing in CSP‖B can be relaxed as follows. If a CSP‖B system with machine sharing in the B part meets the following requirements:

- The CSP system viewed as a B entity together with the B part respects Property 2 (as presented in the previous section)
- The controllers, at least those that involve shared machines, respect Property 3

then the CSP‖B system is consistent for the parts sharing B machines. The rest of the system can be verified e.g. with the techniques of [3].

Discussion about Other Architectural Patterns. The solution for introducing shared B machines in a CSP||B system also gives clues about other kinds of architectural evolutions for a CSP||B system. The "one machine-several controllers" as in Fig. 5(c) is already handled by the consistency definition in [3]. The "one controller-several machines" case illustrated by Fig. 5(b) is conjectured to be solved by our approach. Assuming that the controller does not contain any parallel composition, as is the case usually for CSP controllers, then there is no interference problem. Hence the problem here is strictly reduced to the verification of B modular constraints. In case both controlled machines are imported by the CSP controller, our approach does not allow to decide the (in)consistency of the shared machine.

We are left with the case of Fig. 5(c) when modifications happen for all links. In that case, the basic assumptions of B modularity are obviously not met, hence apart from the full use of consistency checking techniques from [3], one would have to use an extension of B allowing such modularity links. We can surmise that the "invariant ownership" approach of [21] or the "rely-guarantee" approach of [22] would fit. Given that Boulmé concludes that [21, conclusion, third paragraph] the rely-guarantee approach is more modular, we suggest that using Büchi's extension of B as a replacement for classical B would bring what is needed for such an architectural case. As this extension impacts mostly the modularity of B and not its core (set theory and substitutions), we think the changes needed at the level of CSP||B would be minor.

6 Conclusion

This paper proposed a B-based solution for allowing architectures with B state sharing in the CSP||B components. The proposal involved the verification that the shared B machine has not its invariant broken, and that the introduction of sharing does not disturb the components. As the first verification is rooted to B semantics, we proposed a verification methodology based on the fact that the CSP parts of the system can be viewed as a single B machine. We thus were left with characterising the links between CSP controllers and B machines as B modularity links. We have shown that the verification could thus be reduced to check that the B modularity constraints are satisfied.

The second verification involved problems of interference between controllers. We adapted and simplified the solution proposed by Evans & Treharne [3] for verifying the non-interference of controllers. We exploit the additional knowledge given by modularity links at the B level to naturally deduce non-interference properties from the modularity links.

Related Work Addressing Sharing in B and CSP||B. Let us now compare this approach to similar approaches applied to CSP||B or B alone (see Sect. 4).

On the one hand, the architecture of Fig. 5(c) was first introduced in [3], thanks to the use of uniform properties for deciding machine consistency. The reason was that the use of rely-guarantee properties when analysing the consistency of a controlled machine allowed one controller keeping track of what

the other controller could change or not in the machine. In [4], Evans et.al used CSP controllers "augmented" with a B part to perform automatic consistency and non-discrimination checks of CSP||B models of information systems. Our approach deals mostly with the B part, hence it can be viewed as complementary. Those works and ours could thus be used together to bring state sharing at every level of the CSP||B formalism.

On the other hand, several works on the B formalism proposed tightened modularity constraints for ensuring the absence of inconsistency or extending the formalism for allowing some useful kinds of sharing. The already mentioned in Sect. 2.1 works in [8,7] are still situated in the single-writer paradigm. Assuming the CSP controllers can be viewed as a single B entity, the modularity constraints would allow the architectures (a) and (b) of Fig. 5, because of the clear separation of the seeing (read-only) paths and the importing (read-write) paths. These tightened modularity constraints were quickly integrated into the B commercial tools.

A few works have attempted to deal with the multiple-writer paradigm within the B method. Boulmé and Potet [21] proposed an approach inspired by a similar technique of Spec#, where a developer can mark at what places a shared object (hence, for B, a shared machine) can have its invariant broken. This allows having a broader set of architectures for B but the drawback is a greater number of proof obligations. This approach has no tool support we are aware of.

Büchi and Back [22] proposed changing the B modularity mechanisms to allow for multiple writers in a rely-guarantee fashion. B machines become equipped with contracts, each describing several roles. Each contract corresponds to a way of sharing the machine, with all roles corresponding to a way of invoking the operations of the shared machine. In our opinion, only a combination of CSP with Büchi's B along with the use of uniform properties could deal with the architecture of Fig. 5(d), because of multiple-writers at the B level and the danger of interferences at the CSP controllers level.

Butler [19] proposed a way of translating CSP systems into action systems, which was later adapted to the B method [20]. The translation keeps the semantics of the CSP operators (sequence, parallel, interleaving) with the additional following constraints: interleaving can only happen at the outermost level and another constraint relevant to the use of so-called conjoined B machines, which is a peculiarity of *csp2b* that we do not use. Finally, viewing the CSP part of a CSP||B system as a B entity is possible.

Finally, Event-B—an event-based variant of the classical B—does not provide sharing mechanisms, but some extensions propose sharing solutions to augment the scalability of Event-B: parallel (de-)compoition of events/machines and their refinement [23], or modularization like in [24]. In addition, let us note that CSP||Event-B systems have recently been studied wrt. modularity and refinement [25,26]: the deadlock-freeness is ensured under some conditions, and the combined specification refinement guarantees the CSP trace refinement but not the failure refinement.

Perspectives. Our proposal allows the relaxation of some constraints upon B machines in a CSP||B system allowing more flexibility with choosing specification architectures. From there, we conjecture that most architectural patterns can be solved with a combination of our solution and the consistency checking rules of [3]. We think at this point that, for addressing the multiple-writers problem at both the level of CSP||B and B, one would need using another extension of B allowing such a paradigm. A version of B extended with rely-guarantee contracts [22] seems to be a good candidate.

Longer-term perspectives include the study of CSP||B component refinements adapted to our problem. Preliminary studies of recent advances in this domain [27] imply that the kind of refinement we seek would be different because of a more complex evolution of the B part through the design. Other interesting perspectives would involve the adaptation of the consistency rules of [3] from PVS to ProB—an animator and model checker for the B method [28], or to a library for the B method in Coq [29], as the affinity of Coq with fixed-point reasoning could help in the verification of uniform properties.

References

1. Colin, S., Lanoix, A., Kouchnarenko, O., Souquières, J.: Towards Validating a Platoon of Cristal Vehicles Using CSP||B. In: Meseguer, J., Roşu, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 139–144. Springer, Heidelberg (2008)
2. Laleau, R., Semmak, F., Matoussi, A., Petit, D., Hammad, A., Tatibouet, B.: A first attempt to combine sysml requirements diagrams and b. Innovations in Systems and Software Engineering 6, 47–54 (2010)
3. Evans, N., Treharne, H.: Interactive tool support for CSP || B consistency checking. Formal Aspects of Computing 19(3), 277–302 (2007)
4. Evans, N., Treharne, H., Laleau, R., Frappier, M.: Applying csp || b to information systems. Software and System Modeling 7(1), 85–102 (2008)
5. Abrial, J.R.: The B Book - Assigning Programs to Meanings. Cambridge University Press (1996)
6. Badeau, F., Amelot, A.: Using B as a High Level Programming Language in an Industrial Project: Roissy VAL. In: Treharne, H., King, S., Henson, M., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 334–354. Springer, Heidelberg (2005)
7. Rouzaud, Y.: Interpreting the B-Method in the Refinement Calculus. In: Wing, J., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 411–430. Springer, Heidelberg (1999)
8. Potet, M.-L., Rouzaud, Y.: Composition and Refinement in the B-Method. In: Bert, D. (ed.) B 1998. LNCS, vol. 1393, pp. 46–65. Springer, Heidelberg (1998)
9. Roscoe, A.W.: The theory and Practice of Concurrency. Prentice Hall (1997)
10. Schneider, S.A., Treharne, H.E.: CSP theorems for communicating B machines. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004, Special issue of IFM 2004. LNCS, vol. 2999, Springer, Heidelberg (2004)
11. Evans, N., Treharne, H.E.: Investigating a file transfer protocol using CSP and B. Software and Systems Modelling Journal 4, 258–276 (2005)
12. Schneider, S., Cavalcanti, A., Treharne, H., Woodcock, J.: A layered behavioural model of platelets. In: 11th IEEE Int. Conf. on Engieerging of Complex Computer Systems, ICECCS (2006)

13. Treharne, H., Schneider, S.: Using a process algebra to control B OPERATIONS. In: 1st Int. Conf. on Integrated Formal Methods (IFM 1999), pp. 437–457. Springer, York (1999)
14. Schneider, S., Treharne, H.: Communicating B Machines. In: Bert, D., Bowen, J., Henson, M., Robinson, K. (eds.) ZB 2002. LNCS, vol. 2272, pp. 416–435. Springer, Heidelberg (2002)
15. Lanoix, A.: Event-B specification of a situated multi-agent system: Study of a platoon of vehicles. In: 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE), pp. 297–304. IEEE Computer Society (2008)
16. Colin, S., Lanoix, A., Kouchnarenko, O., Souquières, J.: Using CSP||B Components: Application to a Platoon of Vehicles. In: Cofer, D., Fantechi, A. (eds.) FMICS 2008. LNCS, vol. 5596, pp. 103–118. Springer, Heidelberg (2009)
17. van Lamsweerde, A.: Goal-driven requirements engineering: the KAOS approach (2009), http://www.info.ucl.ac.be/~avl/ReqEng.html
18. Clearsy: B language reference manual, v1.8.6 (2007)
19. Butler, M.J.: A CSP Approach To Action Systems. PhD thesis, Oxford (1992)
20. Butler, M.: csp2B: A Practical Approach to Combining CSP and B. In: Wing, J., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 490–508. Springer, Heidelberg (1999)
21. Boulmé, S., Potet, M.-L.: Interpreting Invariant Composition in the B Method Using the Spec# Ownership Relation: A Way to Explain and Relax B Restrictions. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 4–18. Springer, Heidelberg (2007)
22. Büchi, M., Back, R.: Compositional Symmetric Sharing in B. In: Wing, J., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 431–451. Springer, Heidelberg (1999)
23. Butler, M.: Decomposition Structures for Event-B. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 20–38. Springer, Heidelberg (2009)
24. Iliasov, A., Troubitsyna, E., Laibinis, L., Romanovsky, A., Varpaaniemi, K., Ilic, D., Latvala, T.: Supporting Reuse in Event B Development: Modularisation Approach. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 174–188. Springer, Heidelberg (2010)
25. Schneider, S., Treharne, H., Wehrheim, H.: A CSP Approach to Control in Event-B. In: Méry, D., Merz, S. (eds.) IFM 2010. LNCS, vol. 6396, pp. 260–274. Springer, Heidelberg (2010)
26. Schneider, S., Treharne, H., Wehrheim, H.: Bounded retransmission in Event-B||CSP: a case study. Electronic Notes in Theoretical Computer Science 280, 69–80 (2011); Proceedings of the B 2011 Workshop
27. Schneider, S., Treharne, H.: Changing system interfaces consistently: A new refinement strategy for CSP||B. Science of Computer Programming 76(10), 837–860 (2011)
28. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
29. Colin, S., Mariano, G.: BiCoax, a proof tool traceable to the BBook. In: From Research to Teaching Formal Methods - The B Method, TFM B 2009 (2009)

P*a*CE: A Data-Flow Coordination Language for Asynchronous Network-Based Applications

Mauro Caporuscio, Marco Funaro, and Carlo Ghezzi

Politecnico di Milano, Deep-SE Group - Dipartimento di Elettronica e Informazione
Piazza L. da Vinci, 32 - 20133 Milano, Italy
{caporuscio,funaro,ghezzi}@elet.polimi.it

Abstract. Network-based applications usually rely on the explicit distribution of components, which interact by means of message passing. Assembling components into a workflow is challenging due to the asynchronism inherent to the underlying message-passing communication model. This paper presents the P*a*CE language, which aims at coordinating asynchronous network-based components by exploiting the data-flow execution model. Specifically, P*a*CE has been designed for dealing with components compliant with the P-REST architectural style for pervasive adaptive systems. Moreover P*a*CE provides reflective features enabling run-time adaptation and evolution of workflows.

1 Introduction

The advent of new resource-constrained mobile computing devices (e.g., smartphones, and tablets) equipped with wireless networking technologies (e.g., WiFi, Bluetooth and 3G), together with the exploitation of new computing paradigms (e.g., Service Oriented Computing, Cloud Computing, and Pervasive Computing), is boosting a fast move from developing applications as standalone systems, to developing applications as network-based systems. Specifically, *network-based systems* rely on the explicit distribution of components, which interact by means of (asynchronous) message passing. Indeed, *network-based systems* differ from *distributed systems* in the fact that the involved networked components are independent and autonomous, rather than considered as integral part of a conceptually monolithic system [28].

In this settings, network-based applications can be easily modeled and developed as a set of interacting actors [4]. An actor is a computational unit that reacts to external stimuli (e.g., messages) by executing one or more of the following actions when stimulated: (*i*) sending messages to other actors, (*ii*) creating new actors, and (*iii*) designating the behavior for the next stimulus. Since there is no causal sequentiality between these actions, they can be carried on in parallel. Indeed, the Actor model is characterized by inherent concurrency among actors, dynamic creation of actors, and interaction through explicit asynchronous message passing (with no restriction on message arrival order).

T. Gschwind et al. (Eds.): SC 2012, LNCS 7306, pp. 51–67, 2012.

Although Actors are proper abstractions for modeling single reactive components that simply react to external stimuli, asynchronism makes them difficult to deal with when modeling *component compositions*. A composition is an "active" actor, also referred to as *orchestrator*, which orchestrates a process by accessing other components and consuming their artifacts. Indeed, the orchestrator queries actors, and aggregates their responses, to achieve its goal. However, since interactions are asynchronous, the orchestrator does not block its execution while waiting for responses. Rather, the orchestrator continues executing, and responses are processed asynchronously, with no causal order. To cope with such issues, we raised the level of abstraction, and devised a new coordination language satisfying the following requirements:

1. Using a RPC-like syntax
2. Retaining the inherent asynchronism of the pervasive environment
3. Making distribution and code parallelization as seamless as possible
4. Integrating local functions to carry out operations which are not coordination-related (i.e., manipulating the local state)

This paper presents P*a*CE (Prime Coordination languagE), a data-flow language for coordinating asynchronous network-based components. Data-flow languages [21][16] structure applications as a directed graph of autonomous software components that exchange data by asynchronous message passing. In the data-flow paradigm the components do not "call" each other, rather they are activated by the run time system, and react according to the provided input (received message). Once the output is available, the run time system is in charge of moving data towards the proper destination. Data-flow applications are inherently parallel. Exploiting the data-flow paradigm introduces a set of advantages: 1) concurrency and parallelism are natural and components can be easily distributed across the network, 2) asynchronous message passing is natural for coordinating independent and autonomous components, and 3) applications are flexible and extensible since components can be hierarchically composed to create more complex functionalities.

Specifically, the P*a*CE language has been designed and developed for composing and coordinating components built according to the P-REST architectural style [9], where components are called resources (we will use this term from now on) and are first-class abstraction acting as "prosumer" [24] – i.e., fulfilling both roles of producer (reactive actor) and consumer (active actor). To support the P-REST style we implemented the PRIME (P-rest Run-tIME) middleware [10]. Since PRIME has been specifically designed to deal with pervasive environments, where applications must support adaptive and evolutionary situation-aware behaviors, achieving *Adaptation* and *Evolution* is primary requirement for the middleware. *Adaptation* refers to the ability to self-react to environmental changes to keep satisfying the requirements, whereas *evolution* refers to the ability of satisfying new or different requirements. PRIME satisfies such goals by providing support for: (*i*) *flexibility*, the middleware is able to deal with the run-time growth of the application in terms of involved resources, (*ii*) *genericity*, the

middleware accommodate heterogeneous and unforeseen functionalities into the running application, and (*iii*) *dynamism*, the middleware is able to discover new functionality at run time and rearrange the application accordingly.

Therefore, PaCE allows developers to specify the active behavior (composition logic) of a *composite resource* in terms of the set of operations defined by the PRIME programming model. Moreover, PaCE exploits PRIME features to achieve both adaptation and evolution of compositions in terms of *resource addition*, *resource removal*, *resource substitution*, and *resource rewiring* [23].

The paper is organized as follows: Section 2 discusses related work, Section 3 overviews the PRIME middleware, and its programming model. Section 4.1, Section 4.2, and Section 5 present the PaCE syntax, semantics, and interpreter, respectively. Section 5.1 discusses dynamic adaptation features provided by PaCE, and Section 6 assesses the work done by presenting a case study. Finally, Section 7 concludes the paper and sketches our perspectives for future work.

2 Related Work

The growth in complexity and heterogeneity of software systems imposes the need of raising the level of abstraction to make the software development process as rigorous as possible. Gelernter and Carriero [13] advocated for the sharp separation between computation (i.e., the tasks that must be executed to achieve the final goal), and coordination (i.e., how the tasks must be arranged to achieve the final goal) in large systems. PaCE is a data-flow coordination language.

Data-flow languages emphasize *data*, and the transformations applied to it to produce desired outputs. The introduction of this perspective is mainly motivated by the inherent unsuitability of the Von Neumann's architecture to the massive parallelism due to its global program counter and its shared memory that rapidly become bottlenecks in parallel programs [6].

In the data-flow computational model, a program is represented by a directed graph built at compile time, where nodes represent instructions, and arcs represent the data dependencies between instructions. When all the arcs entering a node (the *firing set*) have data on them, the node becomes *fireable*. During the execution, the instruction (represented by the fireable node) is executed and its result is placed on (at least) one of the outgoing arcs. Then, the node suspends executing as long as it is again non-fireable. Figure 1 shows the translation of a simple program (on the left-hand side) into the equivalent data-flow graph (on the right-hand side).

Liskov and Shrira in [19] introduced a similar solution for asynchronous and type-safe RPC in the Argus programming language. The basic idea is similar (i.e., continue executing as long as it is possible), but, with respect to data-flow languages, the degree of parallelism attained is lower because of the benefits granted by functional features.

Therefore, exploiting a data-flow approach to compose and coordinate software components is very appealing and the benefit is twofold: (*i*) focusing on

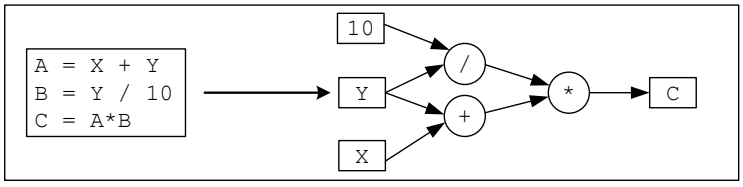


Fig. 1. A simple program and its translation into the equivalent data-flow graph

data allows for a more natural composition modeling approach since, if aided with a visual support, it can also be used and understood by non technical people; (*ii*) developers do not explicitly care about tasks concurrency. Rather, the execution model allows for automatic parallelization.

Pautasso and Alonso exploited the former benefit by proposing the JOpera visual composition language [26], and run-time support [25]. The JOpera language models both data-flow and control-flow dependencies among the tasks in the composition and the development environment is in charge of keeping the two perspectives consistent. The approach does not exploit the data-flow model to implicitly achieve parallelization, rather, the latter is achieved through imperative constructs inserted in the control-flow perspective. In [26] Pautasso and Alonso point out that the data-flow perspective is not enough to model every process because it ignores indirect dependencies (e.g., tasks communicating through databases or configuration files) or because there are dependencies that are not data-related like a compensation handler. P*a*CE addresses the first issue by adopting a purely functional approach, where no side-effects are allowed, and thus no indirect dependencies can be introduced. On the other hand, the compensation handler issue is out of our research scope.

Regarding the latter benefit – i.e., implicit parallelization – a complete general-purpose coordination language has not been proposed. Rather, researchers focused on raising the level of abstraction by proposing languages where nodes in the data-flow graph are functions written in different languages. For example Bernini and Mosconi [8] proposed a visual data-flow language called VIPERS where the node in the data-flow graph are Tcl fragments. Also textual approaches exist, like GLU [14] that embeds C fragments in the LUCID [29] data-flow language. These solutions exploit the implicit parallelism and delegate to other languages the whole computation. As a final remark, they are mostly focused on exploiting parallel computers for scientific computations and are not designed to fit distributed environments.

Another research area slightly related to this work, comprises agent-based workflow modeling and enactment. Indeed, an agent is an autonomous and intelligent program that make decisions on next actions to perform based on its current state. Hence, in agent-based workflow management, agents are provided with goals extracted from the overall workflow schema, thus each of them develops and its own work plan to achieve those goals. Agents-based workflow management systems coordinate agents by exploiting different approaches [15]:

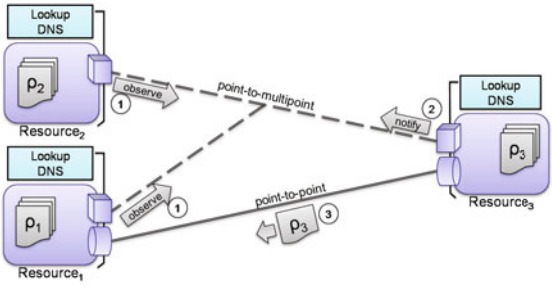


Fig. 2. P-REST architectural style

(*i*) *role-based*, where different agents fulfill different roles and perform a workflow autonomously, (*ii*) *activity-based*, where agents coordinate the execution of activities as defined within the workflow schema without the need for a central workflow enactment service, and (*iii*) *mobility-based*, where the workflow instance is migrated to different locations to perform specific tasks.

3 The P-REST Approach at a Glance

In this section we briefly introduce the Pervasive-REST (P-REST) [9] architectural style, and the PRIME middleware, which provides the run-time support needed for implementing P-RESTful applications – i.e., applications built following the P-REST style.

The P-REST architectural style (depicted in Figure 2) is defined as a refinement of the well known REST architectural style [12], to specifically deal with pervasive environments. P-REST promotes the use of *Resource* as first-class object that plays the role of "prosumer" [24], i.e., an entity that fulfills both roles of provider and consumer. To support coordination among resources, P-REST extends the traditional request/response mechanism through new primitives: (*i*) a *Lookup* service that enables the discovery of new resources at run time, (*ii*) a distributed *Domain Name System* (DNS) [22] service that maintains the mappings between resource URIs and their actual location in case of mobility, and (*iii*) a coordination model based on the Observer pattern [18] that allows a resource to express its interest in a given resource and to be notified whenever changes occur.

In P-REST, resources directly interact with each other by exchanging their representations. Referring to Figure 2, both Resource_1 and Resource_2 observe Resource_3 (messages ①). When a change occurs in Resource_3, it notifies (message ②) the observer resources. As the notification is received, Resource_1 issues a request for Resource_3 and obtains as a result the representation ρ_3 (message ③). Note that, while observe/notify interactions take place through the *point-to-multipoint* connector (represented as a cube), REST operations exploit *point-to-point* connector (represented as a cylinder). All the resources exploit both

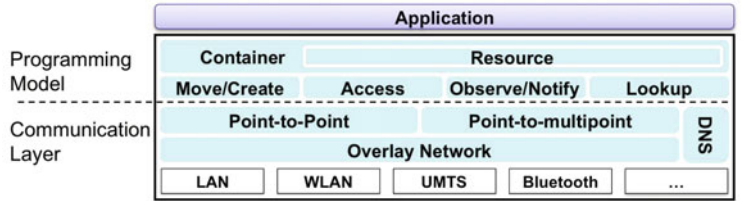


Fig. 3. PRIME layers

the `Lookup` operation to discover the needed resources, and the DNS service to translate URIs into physical addresses.

According to the *uniformity* principle [12] P-REST describes every software artifact as a Resource implementing a set of well-defined operations, namely `PUT`, `DELETE`, `POST`, `GET`, and `INSPECT`. Moreover, P-REST adopts semantic resource's descriptions to specify both functional and non-functional properties of a resource. Indeed, descriptions support the implementation of the lookup service by enabling run-time semantic-aware resource discovery.

P-REST enhances the REST *addressability* principle – i.e., a resources is identified by means of an URI – by distinguishing between *Concrete URI* (CURI) and *Abstract URI* (AURI). CURI identify concrete resource instances, whereas AURI identify groups of resources. Such groups are formed by imposing constraints on resource descriptions (e.g., all the resources implementing the same functionality). Therefore, CURI achieves point-to-point communication, and AURI achieves point-to-multipoint communication. Resources can be used as building-blocks for composing complex functionalities. A *Composition* is still a resource that can, in turn, be used as a building-block by another compositions. Resources involved in a composition are handled by means of a *Composition Logic*.

The PRIME(P-rest Run-tIME) [10] middleware provides the run-time support for the development of P-RESTful applications[1]. Referring to Figure 3, the PRIME architecture exploits a two-layer design where each layer deals with a specific issue:

Communication layer – To deal with the inherent instability of pervasive environments, PRIME arranges devices in an overlay network built on top of low-level wireless communication technologies (e.g., Bluetooth, Wi-Fi, Wi-Fi Direct, and UMTS). Such an overlay is then exploited to provide two basic communication facilities, namely *point-to-point* and *point-to-multipoint*. *Point-to-point* communication grants a given node direct access to a remote node, whereas *point-to-multipoint* communication allows a given node to interact with many different nodes at the same time. Furthermore, the PRIME communication layer provides facilities for managing both physical and logical mobility [27].

[1] PRIME is available at `http://code.google.com/p/prime-middleware/`, under the GNU/GPLv3 license.

Programming model – PRIME provides the programming abstractions to implement P-RESTful applications by leveraging the functional programming features of the Scala language [1] and the Actor Model [4]. In particular, PRIME defines two main abstractions and a set of operations to be performed on them. *Resource* represents the computation unit, whereas *Container* handles both the life-cycle and the provision of resources. The set of operations allowed on resources defines the message-based PRIME *interaction protocol* and includes: (*i*) *Access*, which gathers the set of messages to access and manipulate resources, (*ii*) *Observe/Notify*, which allows resources to declare interest in a given resource and to be notified whenever changes occur, (*iii*) *Create*, which provides the mechanism for creating a new resource at a given location, and *Move*, which provides the mechanism to relocate an existing resource to a new location, and (*iv*) *Lookup*, which allows for discovering new resources on the basis of a given semantics-aware description.

It is worth noticing that the PRIME programming model exploits the Actor Model, which in turn relies on the PRIME communication layer to provide message-passing interaction among actors.

4 P*a*CE – Prime Coordination languagE

This section presents both the syntax (§4.1) and semantics (§4.2) of P*a*CE, which have been specifically designed to offer the set of core features characterizing data-flow languages [16]: (1) single assignment of variables, (2) freedom from side-effects, (3) data dependencies equivalent to scheduling (statements are not executed in the order they are written, but as their input data become available), (4) an unusual notation for iterations (due to features 1 and 2), and (5) lack of history sensitivity in procedures (in a language without a deterministic control flow, histories cannot be univocally built by a developer).

Realizing such features strongly impacts on both syntax and semantics. In fact, (1) and (2) ask for a functional programming style, where multiple variable assignment is avoided, and functions are side-effects free (i.e., do not affect the environment, and their results depend only on input values). Moreover, features (1) and (2) are fundamental to induce scheduling from data dependencies (3). In fact, since scheduling is determined from data dependencies, it is important to guarantee that variables do not change between their definition and their use. Whenever variables are modified at run time, the data-flow graph (see Figure 1) would be invalidated. On the other hand, due to single-assignment, the order of statements is in general not relevant. However, single assignment conflicts with the imperative style in loops (4) because it forbids the increment of loop variables, thus loops are implemented through special constructs. Finally, data-flow languages inherit from the functional languages the lack of history sensitivity for procedures (5). In a language without a deterministic order of execution, histories cannot be univocally built by a programmer. Therefore, operations rely on the input parameters only and not on previous invocations. The functional operators along with the ordered queues are enough to guarantee a deterministic

```
ID          = [A-Za-z]([A-Za-z] | [0-9])*
INTEGER     = [1-9]([0-9])*
STRING      = "([A-Za-z] | [0-9])*"
URI         = ID | STRING
OP          = OPTWOPAR '(' URI ',' ID ')'
            | OPONENOPAR '(' URI ')'
OPTWOPAR    = 'post' | 'put'
OPONEPAR    = 'get' | 'delete' | 'inspect'
BLOCK       = '{' STATEMENT+ '}'
STATEMENT   = LOOP | ASSGNM | OBSERVE | CREATE | OUTPUT
            | INFLOOP | IF | WRITE | LOOKUP | SMFUN
OBSERVE     = 'observe' '(' URI ')' BLOCK
CREATE      = ID '=' 'create' '(' URI ',' ID ')'
            | ID '=' 'create' '(' URI ',' ID ',' URI ')'
LOOKUP      = ID '=' 'lookup(' ID ')'
ASSGNM      = ID '=' OP
            | ID '=' ID '(' (STRING)? (',' STRING)* ')'
            | ID '=' 'get(' 'stdin' ')'
SMFUN       ID '(' (STRING)? (',' STRING)* ')'
OUTPUT      = ID '(' (STRING)? (',' STRING)* ')'
            | 'put' '(' 'stdout' ',' ID ')'
            | 'put' '(' 'stdout' ',' STRING ')'
LOOP        = 'while' ID 'in' ID 'to' ID BLOCK
INFLOOP     = 'while' '(' 'true' ')' BLOCK
IF          = 'if' '(' BOOLEXP ')' BLOCK 'else' BLOCK
BOOLEXP     = BOOLEXP
            ( '&&' | '||' | '<' | '>'| '<=' | '>=' | '==' )
            BOOLEXP
            | ID | INTEGER | STRING |'!' BOOLEXP
            | '(' BOOLEXP ')'| 'true'| 'false'
```

Fig. 4. EBNF for P*a*CE

behavior for the model, that is, for a given set of inputs, a program always produces the same set of outputs [5][11][17].

4.1 Syntax

According to the above guidelines, P*a*CE's syntax is mainly inspired by common functional languages but for control structures, which instead are close to the imperative style. Therefore, every instruction is an assignment, operations are side-effects free (i.e., do not affect the environment, and their results depend only on input values), and multiple variable assignment is avoided. Moreover, since P*a*CE is tailored to the P-REST style, it directly embeds P-REST operations, which in turn are straightforwardly mapped to the PRIME programming model.

Referring to the generative EBNF for P*a*CE (see Figure 4), PRIME's access operations are derived from the `OP` non-terminal. `OPONEPAR` operations are invoked with one mandatory parameter – i.e., the list of `URI`s identifying the target

resources –, whereas `OPTWOPAR` operations get the `URI`s list, and an additional parameter containing a representation. Return values are lists of representations, and depend on the specific operation used: `GET` returns the representation of the target resource; `PUT`, `DELETE` and `POST` return a representation of the *status code* (e.g, "ERROR", "OK", and "NORESPONSE"), and `INSPECT` returns a representation of the resource description. Note that all PaCE operations are designed to work on lists. It is also possible to issue `GET` and `PUT` on two special `URI`s: `stdin` and `stdout`, respectively. The operation `a = GET(stdin)` reads a from the standard input, and assigns a value to the variable `a`. Conversely, the instruction `PUT(stdout, ID)` writes the value of `ID` on the `stdout`.

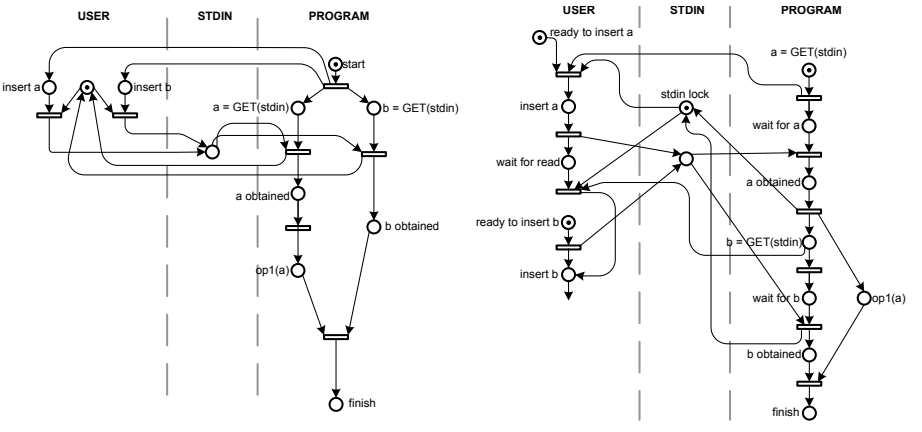
The EBNF can also generate invocation to two categories of external functions. The first one gathers the functions without a return type. These functions, thus, cannot directly affect the PaCE script but can only update the state of the composite resource the script is attached to. For this reason they are called *State-Manipulating Functions* (SMFUN from now on). The second one comprises the functions with a return value that are generated by the `ASSGNM` nonterminal. Notice that also these functions might have side effects on the composite resource state.

The `LOOKUP` operation is invoked with one parameter, which represents the identifier of the external function used to filter out the resources, and returns a list of URIs. The `CREATE` operation gets as input the container `URI` where the resource has to be created, and the representation used to initialize the new resource. A third parameter can be provided to impose a specific `URI` for the new resource. The return value is the `URI` of the new resource. The `OBSERVE` operation, which exploits the event-driven communication model provided by PRIME, is defined as control structure. `OBSERVE` defines a block of statements that is executed whenever an event, generated by observed resources (specified by the `URI` parameter), is received.

PaCE provides a set of simple control structures. The infinite loop (i.e., **while** `(true) {...}`), and conditional structure (i.e., **if** `(cond) {...}` **else** `{...}`) have the usual syntax. Whereas the `LOOP` control structure requires for special attention. As already mentioned, the *single assignment* feature prevents the implementation of standard loops where the loop variable is explicitly incremented at every iteration. To overcome such an issue, PaCE defines a control structure of the form: **while** `var` **in** $\texttt{b}_1$ **to** $\texttt{b}_2$ `{...}`, where `var` ranges from $\texttt{b}_1$ to $\texttt{b}_2$ by preventing its explicit assignment within the loop.

4.2 Semantics

Since PaCE is inspired by data-flow languages, it adheres to the data-flow execution model. However, while data-flow languages build explicitly the data-flow graph to drive the execution, in PaCE such graph is built implicitly. Indeed, PaCE does not completely depart from the sequential execution of instructions, but makes it asynchronous: statements are evaluated sequentially, but their execution is non-blocking. That is, given a two-statements sequence $< S_1, S_2 >$, S_2 can be executed independently of the S_1 termination, as long as the



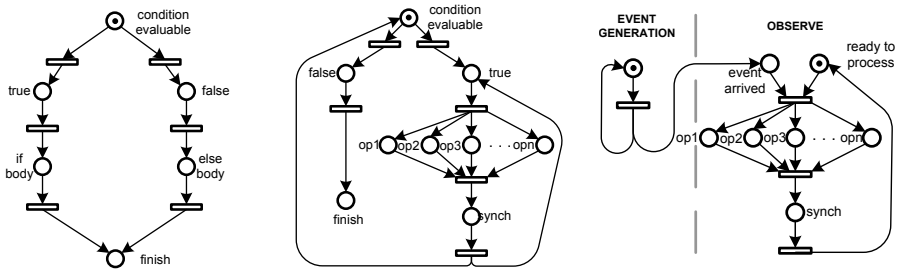
(a) Pure data-flow execution model (b) P*a*CE execution model

Fig. 5. Execution model

execution of S_2 does not need data produced by S_1. When a statement is invoked, it returns immediately by yielding a *future* variable – i.e., a special variable that will eventually contain the result of an asynchronous computation. Whenever the variable is accessed, execution is suspended if the value is not available, yet.

P*a*CE scripts are compliant with the data-flow execution model. However, their execution is not purely concurrent since parallel operations are executed as soon as the interpreter evaluates them, and needed data is available. Having non-pure parallelism, allows the P*a*CE interpreter to retain at run time the information about the instruction order. This is particularly important when dealing with I/O and control structures. In the data-flow execution model, sequentiality of instructions is lost when the data-flow graph is built, since only data dependencies are considered. For instance, let S_1 =<`a = get(stdin), c = op1(a) b = get(stdin)`> be a sequence with two independent `get`s. According to the data-flow execution model, they can be executed in any order. As a consequence, it is impossible to univocally determine which result, coming from a `get(stdin)`, must be stored in `a`, and which in `b` (see Figure 5a). To this end, P*a*CE semantics imposes the mutually-exclusive access to the `stdin` resource by exploiting the information about the instruction order. When the first `get(stdin)` is executed, it locks the standard input; then, when the second `get` tries to access `stdin`, it is suspended as long as the first `get` completes. In this case the two `get`s are ordered, and `stdin lock` ensures the mutual-exclusion policy for accessing to the standard input. It is worth noticing that such an issues does not concern the `put(stdout, ID)`, which can be executed whenever the `ID` variable is available.

As introduced in Section 4.1, P*a*CE scripts can also contain control structures: i.e., conditional (`if-else`), loops (`while`), and `observe` structures. Notice that for the sake of clarity, the Petri nets in Figure 6 do not take into account possible dependencies between instructions in the control structures's bodies and



(a) `if` (b) `while` (c) `observe`

Fig. 6. Semantics of the P*a*CE control structures

instructions outside the bodies. However, whenever any dependency holds, it is treated according to the data-flow execution model.

Conditional Structure – According to the execution model discussed above, the conditional structure is evaluated as soon as the *conditional expression* becomes evaluable. Hence, the proper branch is executed. Figure 6a shows the Petri net specifying the `if-else` semantics.

Loop Structure – As introduced in Section 4.1, the concept of loop does not fit the data-flow paradigm. However, having loops instead of the equivalent tail recursion, is fundamental for the adoption of the paradigm [3]. Therefore, the rationale of the loop syntax (see Figure 4) relies in the fact that P*a*CE is conceived as a coordination language. Indeed, any computation should be accomplished by either remote resources or external functions. According to the syntax given above, every loop iteration is forced to happen in isolation. That is, all the instructions in the loop body must complete before the next iteration can be executed. Moreover, all the variables, allocated within an iteration, are deallocated at the end the iteration, to guarantee the single-assignment property. Data dependences holding between instructions in the loop body and external instructions must be satisfied before the execution of the first iteration. Figure 6b shows the Petri net defining the `while` semantics. Note that, the isolation is guaranteed by `synch`, which forces the Petri net to wait until all the instructions in the body complete.

Observe Structure – The `observe` operation introduces the event-based programming paradigm: the `observe` is an infinite loop whose body is executed every time an event is produced in one of the observed resources. To avoid locking the execution in such infinite loops, every `observe` is executed separately to allow the interpretation of a script to continue beyond any `observe`. According to P-REST (see Section 3), the `observe` operation accepts as input the list of resources to be observed (specified by means of their `URI`s). Whenever an event is received from an observed resource, the body is executed according to the

data-flow execution model. Notice that also in this case the body is executed in isolation and events are queued and consumed sequentially. Figure 6c shows the Petri net specifying the `observe` semantics: `event generator` produces tokens (events), which `observe` consumes; `event arrived` models the incoming queue.

External Functions – P*a*CE allows for the definition of two types of external functions: *side-effect-free* and *state-manipulation* functions. *Side-effect-free* functions are used to manipulate P*a*CE variables (e.g., to translate the data from one encoding to another). *State-manipulation* (SMFUN) functions are used to manipulate the internal state of the composite resource. Due to the shared-nothing paradigm exploited by P*a*CE, SMFUN functions are critical, and two main problems arise. First, SMFUNs can conflict with the `POST` operation since it can modify the internal state. Second, concurrent SMFUNs can potentially modify the same data. To this extent, on one hand, P*a*CE provides mutual exclusion mechanisms to avoid the simultaneous access to the state of the composite resource. On the other hand, `POST` and SMFUNs should not be used at the same time to avoid unforeseen (and unpredictable) behaviors of the P*a*CE scripts. A POST changes the internal state of a resource, and SMFUNs can be used to read such internal state. Thus, the script behavior could be implicitly modified, and this fact would break the functional assumption of data-flow languages. Such interactions could lead to unforeseen (and unpredictable) behaviors of the P*a*CE scripts.

5 P*a*CE: Interpreter

This section presents the P*a*CE interpreter, and details how P*a*CE scripts are mapped to the underlying PRIME middleware.

The P*a*CE interpreter is developed in Scala [1], and exploits the Scala parser combinator library [20]. However, since in P*a*CE all the variables are stored as *future variables*, the interpretation algorithm makes use of an auxiliary symbol table containing all the bindings between variable names and future values. According to P*a*CE semantics (§4.2), every operation immediately returns a future variable that will be eventually filled with the result. Whenever the operation to be executed requires input parameters, whose values are not yet available, the interpreter suspends the execution of the analyzed instruction that will be resumed whenever the missing values will become available. Clearly, suspending the execution of an instruction does not suspend the execution of the whole script. Rather, the execution flow proceeds according to the data-flow paradigm, and allows for the implicit construction of the data-flow graph.

The interpreter also implement a basic error-handling mechanism. Requests are automatically reissued if a configurable timeout expires up to three times before terminating the script. This simple mechanism accounts for network problems only. Conversely, errors at the application level (i.e., errors generated by the queried resources) must be taken care of explicitly in the script because they are directly notified in the response.

PaCE implements P-REST operations by exploiting the PRIME middleware, which is in charge of dispatching requests and responses (see Section 3). Since PRIME operations are completely asynchronous, the PaCE interpreter implements an *indexing system* that (*i*) binds received responses to issued requests and, (*ii*) assigns the values embedded in the responses to the proper future variable in the symbol table.

The PaCE interpreter also offers an abstraction for developing and using external functions. In order to be used, external functions must be made available to the interpreter at run time. External functions are defined by extending the `ExternalFunctions` abstract class, which implements all the mechanisms needed to parse, validate and publish functions. `ExternalFunctions` exploits the Java reflection mechanism, and exposes a method `invoke`, which takes as input the name of the functions to be executed and a list (possibly empty) of parameters. The `invoke` method executes the function in a future block to guarantee asynchronism. In addition, `invoke` checks whether the function is a SMFUN or not. In fact, according to the PaCE semantics (§4.2), SMFUN functions are executed in mutual exclusion, whereas non-SMFUN functions can be executed concurrently.

5.1 Run-Time Adaptation

A primary requirement for P-RESTful applications is to support adaptive and evolutionary situation-aware behaviors [9]. To this extent, PaCE provides four operations on resources – i.e., *resource addition*, *resource removal*, *resource substitution*, and *resource rewiring* – that allow PaCE's scripts to be reconfigured at run time.

According to the PaCE syntax and semantics discussed above, *resource addition/removal* simply refer to the ability of adding/removing a `URI` to/from a list, and *resource rewiring* means changing the value of a variable containing a `URI`. *resource substitution*, instead, consists of rewiring a resource binding, and moving the state of the old resource to the new one. It is worth to notice that, to avoid inconsistencies in the symbol table, the interpreter suspends the execution of scripts before performing any reconfiguration. Indeed, reconfigurations are performed asynchronously and in isolation – i.e., reconfigurations are queued and executed only when the interpreter is in a safe state. While *add*, *remove* and *rewire* operations are entirely implemented within the PaCE interpreter, *substitute* exploits the primitives provided by PRIME to retrieve the state of the old resource (`GET`), and to initialize the new one (`PUT`).

Special attention should be paid whenever the adaptation involves a variable containing observed `URI`s. Indeed, observed variables are referred once, at the beginning of the `observe` structure, when the variable is defined, and the corresponding subscription is generated. Hence, adaptation operations must be carefully examined before their application:

add: the intended semantics concerns the addition of a new resource to the pool of resources already observed. Whether an add is performed, a new subscription is issued to start observing the new resource.

remove: the intended semantics is the opposite of the add operation, i.e., the given resource `URI` should be no longer observed. Therefore, the old subscription is removed so that no further notifications will arrive from a specific URI.

rewire: the intended semantics concerns the substitution of the entire resource pool with a new one. This operation is implemented by removing every `URI` from the list, then adding new `URI` to the list.

substitute: this operation, if applied to the variable containing the observed resources, does not affect the behavior of the `observe` block.

6 Case Study

In this section we want to show how P*a*CE can be used to orchestrate PRIME resources. This section also covers the run-time adaptation facilities of the P*a*CE interpreter.

Let us introduce the following Pervasive Slide Show (PSS) scenario: *Carl, a university professor, is going to give a talk in a conference room, and carries his laptop storing both the slides and related handouts. The conference room provides speakers with a smart-screen available on the local wireless network, whereas the audience is supposed to be equipped with devices (e.g., laptops, smartphones and tablets), which can be used for displaying either the currently-projected slide on the screen or the related handouts. The audience and the speaker always refer to the same slide, and to the same page of the handouts. Every device is required to have a running* PRIME *instance.*

The PSS implementation conforms to the P-REST conceptual model and specifies the following resources: `CurrentSlide` and `CurrentPage` represent the currently-projected slide and the corresponding handout page, respectively. `Presentation` is the composition that implements the interface Carl uses to browse the slide-show. It also encapsulates the slides and the handouts along with the pointers that keep track of the current slide and handout page. `Reader` visualizes the slide show or the handout on the audience's devices. `Projector` handles the smart-screen of the conference room. The `Projector` resource is deployed on the smart-screen PRIME container. The other resources are initially deployed on Carl's container, and made available to the devices in the audience which join the slide-show.

When a participant (say, Bob) enters the conference room, he uses the PRIME *resource finder* built-in tool, which lists all of the resources available within the overlay, to explore the environment and find the `Reader` resource. Hence, selecting `Reader` from the list, the PRIME node issues a `GET` operation to retrieve a representation of `Reader`, which, in turn, is used to create a local instance. Figure 7 shows the P*a*CE script for `Presentation` and `Reader`. The `Presentation` resource is a composition meant to aid Carl to project his slides. Thus, the associated P*a*CE script (see Figure 7a) searches the conference center for a projector by using the external function `projSearch`. The resulting URIs are stored in the `proj` variable to be used later. Then, the execution enters in an infinite loop

```
proj = lookup(projSearch)
while (true){
  cmd = GET(stdin)
  if (cmd == ''fwd''){
    rep = getNextSlide()
    PUT(currSlide, rep)
    PUT(proj, rep)
  }
  if (cmd == ''bwd''){
    rep = getPreviousSlide()
    PUT(currSlide, rep)
    PUT(proj, rep)
  }
}
```

(a) Script for `Presentation`

```
observe(slURI){
  slide = GET(obsURI)
  view(slide)
}

while(true){
  cmd = GET(stdin)

  if(cmd == ''ho'')
    rewire(slURI, hoURI)

  if(cmd == ''pres'')
    rewire(slURI, presURI)
}
```

(b) Script for `Reader`

Fig. 7. Scripts for PSS scenario

to serve Carl's commands. In case of a "fwd" ("bwd") command the script calls an SMFUN called `getNextSlide`. It returns the representation of the new slide and, as a side-effect, updates the pointers to the current slide and to the current handout page in the `Presentation` resource. The new representation is stored in the `rep` variable and it is used as a parameter for the following two `PUT` operations: one, issued towards the `CurrSlide` resource, the other, towards the `proj` variable, that is, towards all the the smart-screens found in the conference room. All the `Projector` resources will react by projecting the new slide. As for the `CurrentSlide` resource, it is involved in a more complex interaction. Indeed, looking at Figure 7b, the `Reader` script observes the state of `CurrentSlide` and, whenever it changes, the `Reader` issues a `GET` towards it. When the new slide is retrieved, it is visualized through the `view` external function. Apart from the `observe` body, the script for `Reader` also features an infinite cycle to allow Bob to toggle between the presentation and the handout. This second part makes use of the adaptation primitives described in 5.1. They are used in a reflective way, that is, the script for the `Reader` reconfigures itself to serve Bob's commands. Specifically, when Bob issues the "ho" command, the script invokes the `rewire` functions with `slURI` (the observed URI) as first parameter (the old binding) and the URI of the `CurrentPage` resource (`hoURI`) as second parameter. The `hoURI` variable is hard-wired in the script by the system developer. Being `slURI` an observed variable the special rules presented in 5.1 apply. Specifically, the symbol table is updated so that the `slURI` will denote the `CurrentPage` URI and not the `CurrentSlide` anymore. As a consequence, also the `obsURI` variable is updated accordingly. To make the update effective the `observe` body must be executed once before starting processing the new data stream. After the `rewire` has completed, Bob will be able to follow the handout on his device.

Later on, Bob will be able to switch back to the presentation by issuing the "pres" command.

7 Conclusion

In this paper we addressed the problem of coordinating resources adhering to the P-REST architectural style. Such resources are modeled by PRIME as actors, i.e., an autonomous and asynchronous computational resource reacting to external stimuli. Although Actors are proper abstractions for modeling P-REST resources, asynchronism makes them difficult to deal with when assembled into a workflow. Indeed, resource coordination is a time-consuming and error-prone process for a developer.

To address these problems, in this paper we propose P*a*CE, a data-flow language for composing and coordinating resources built on top of the PRIME middleware [9]. Moreover, P*a*CE exploits PRIME features to enable both run-time adaptation and evolution of compositions. We described P*a*CE's syntax and semantics, and discussed the advantages and issues induced by the adoption of the data-flow paradigm. We also presented the P*a*CE interpreter, which provides reflective capabilities to achieve reconfiguration operations, namely *resource addition*, *resource removal*, *resource substitution*, and *resource rewiring*.

As for the future research directions, we want to improve the language by adding a full support to the error-handling before removing the strict binding between P*a*CE and P-REST, to obtain a general purpose coordination language suitable for every inherently parallel and asynchronous environment. Furthermore, following well known approaches, e.g., Yahoo Pipes [2], Mashlight [7] and JOpera [26], we plan to develop a tool for visually specifying resource compositions. This would further ease the development process, by allowing developers to fully benefit from data-flow paradigm: focus on how things interact, rather than on how things happen.

Acknowledgements. This research has been funded by the European Commission, Programme IDEAS-ERC, Project 227077-SMScom (`http://www.erc-smscom.org`).

References

1. The Scala programming language, `http://www.scala-lang.org/`
2. Yahoo pipes, `http://pipes.yahoo.com/`
3. Ackerman, W.: Data flow languages. Computer 15(2), 15–25 (1982)
4. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge (1986)
5. Arvind, Culler, D.E.: Dataflow architectures. Annual Review of Computer Science 1(1), 225–253 (1986)
6. Backus, J.: Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. Commun. ACM 21, 613–641 (1978)
7. Baresi, L., Guinea, S.: Mashups with Mashlight. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 711–712. Springer, Heidelberg (2010)

8. Bernini, M., Mosconi, M.: Vipers: a data flow visual programming environment based on the tcl language. In: AVI: Proceedings of the Workshop on Advanced Visual Interfaces. Association for Computing Machinery, Inc., New York (1994)
9. Caporuscio, M., Funaro, M., Ghezzi, C.: Restful service architectures for pervasive networking environments. In: Wilde, E., Pautasso, C. (eds.) REST: From Research to Practice. Springer (2011)
10. Caporuscio, M., Funaro, M., Ghezzi, C.: Resource-oriented middleware abstractions for pervasive computing. In: IEEE International Conference on Software Science, Technology and Engineering, SWSTE (to appear, June 2012)
11. Davis, A., Keller, R.: Data flow program graphs. Computer 15(2), 26–41 (1982)
12. Fielding, R.T.: REST: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis. University of California, Irvine (2000)
13. Gelernter, D., Carriero, N.: Coordination languages and their significance. Commun. ACM 35, 97–107 (1992)
14. Jagannathan, R.: Coarse-grain dataflow programming of conventional parallel computers. In: Advanced Topics in Dataflow Computing and Multithreading, pp. 113–129. IEEE Computer Society Press (1995)
15. Joeris, G.: Decentralized and flexible workflow enactment based on task coordination agents. In: 2nd International BiConference Workshop on Agent-Oriented Information Systems (AOIS 2000 - CAiSE 2000), pp. 41–62 (2000)
16. Johnston, W.M., Hanna, J.R.P., Millar, R.J.: Advances in dataflow programming languages. ACM Comput. Surv. 36, 1–34 (2004)
17. Kahn, G.: The Semantics of a Simple Language for Parallel Programming. In: Rosenfeld, J.L. (ed.) Information Processing 1974: Proceedings of the IFIP Congress, pp. 471–475. North-Holland, New York (1974)
18. Khare, R., Taylor, R.N.: Extending the representational state transfer (rest) architectural style for decentralized systems. In: ICSE 2004, pp. 428–437. IEEE Computer Society, Washington, DC (2004)
19. Liskov, B., Shrira, L.: Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, PLDI 1988, pp. 260–267. ACM, New York (1988)
20. Moors, A., Piessens, F., Odersky, M.: Parser combinators in scala. CW Reports, vol. CW491. Department of Computer Science, KU Leuven (2008)
21. Morrison, J.P.: Flow-Based Programming: A New Approach to Application Development, 2nd edn. CreateSpace (2010)
22. Network Working Group. Role of the Domain Name System (DNS). RFC3467
23. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: ICSE 1998 (1998)
24. Papadimitriou, D.: Future internet - the cross-etp vision document, version 1.0 (January 2009), http://www.future-internet.eu
25. Pautasso, C., Alonso, G.: Jopera: A toolkit for efficient visual composition of web services. Int. J. Electron. Commerce 9, 107–141 (2005)
26. Pautasso, C., Alonso, G.: The jopera visual composition language. Journal of Visual Languages & Computing 16(1-2), 119–152 (2005); 2003 IEEE Symposium on Human Centric Computing Languages and Environments
27. Roman, G.-C., Picco, G.P., Murphy, A.L.: Software engineering for mobility: a roadmap. In: FOSE 2000, pp. 241–258. ACM, New York (2000)
28. Tanenbaum, A.S., Van Renesse, R.: Distributed operating systems. ACM Comput. Surv. 17, 419–470 (1985)
29. Wadge, W.W., Ashcroft, E.A.: LUCID, the dataflow programming language. Academic Press Professional, Inc., San Diego (1985)

Adaptation of Legacy Codes to Context-Aware Composition Using Aspect-Oriented Programming

Antonina Danylenko and Welf Löwe

Linnaeus University, Software Technology Group,
351 95 Växjö, Sweden
{antonina.danylenko,welf.lowe}@lnu.se

Abstract. The context-aware composition approach (CAC) has shown to improve the performance of object-oriented applications on modern multi-core hardware by selecting between different (sequential and parallel) component variants in different (call and hardware) contexts. However, introducing CAC in legacy applications can be time-consuming and requires quite some effort for changing and adapting the existing code. We observe that CAC-concerns, like offline component variant profiling and runtime selection of the champion variant, can be separated from the legacy application code. We suggest separating and reusing these CAC concerns when introducing CAC to different legacy applications.

For automating this process, we propose an approach based on Aspect-Oriented Programming (AOP) and Reflective Programming. It shows that manual adaptation to CAC requires more programming than the AOP-based approach; almost three times in our experiments. Moreover, the AOP-based approach speeds up the execution time of the legacy code, in our experiments by factors of up to 2.3 and 3.4 on multi-core machines with two and eight cores, respectively. The AOP based approach only introduces a small runtime overhead compared to the manually optimized CAC approach. For different problems, this overhead is about 2-9% of the manual adaptation approach.

These results suggest that AOP-based adaptation can effectively adapt legacy applications to CAC which makes them running efficiently even on multi-core machines.

Keywords: Context-Aware Composition, Autotuning, Aspect-Oriented Programming.

1 Introduction

Context-aware computation is an essential part of a wide range of application domains, where an application should adapt behavior according to potentially changing context or environment during its execution. *Context-oriented programming* is a technique for the design of such applications [1,2]. *Context-aware composition* (CAC) as a special case of context-aware computation aims at adapting applications to changing call contexts and available resources in the system environment. Its goal is to improve performance (or

T. Gschwind et al. (Eds.): SC 2012, LNCS 7306, pp. 68–85, 2012.

other optimization goals) of such applications. Among others, CAC allows to develop performance-portable programs for modern multi-core hardware.

CAC applications evaluate their call contexts and system environment at runtime and, depending on that, select between different alternative sequential and parallel component variants. The selected decisions are based on previous experience from offline profile execution or from online execution monitoring abstracted and aggregated using automated machine learning.

In order to exploit the benefits of the recent hardware development toward multi-core processors, possibly supported by hardware accelerators such as GPUs, applications have to adapt to their respective system environment. Creating new CAC applications is well understood [3,4]. However, introducing CAC in existing applications usually requires a high re-engineering and implementation effort and, therefore, can be time-consuming and error-prone.

Regardless of the concrete applications, adoption of CAC needs to address some common concerns: offline profiling and/or runtime monitoring, machine learning to extrapolate or interpolate the profiling/moitoring results, and runtime variant selection and dynamic composition for each call context and system environment. These concerns can be separated from the actual application.

It has been shown before that context-awareness is in principle a crosscutting concern and, therefore, can be treated as an *aspect* [5]. By means of a so-called *advice*, aspects can be applied to certain program points called *join points*. This can be controlled by the actual context of the join point, its program state, and the state of the system environment, which has to be determined at runtime.

In this paper, we suggest an approach based on Aspect-Oriented (AOP) and Reflective Programming that separates the concerns of offline profiling and runtime composition and reuses them when adapting legacy applications to CAC. The context-awareness aspect becomes reusable regardless of the legacy application. It selects the optimum component variant dynamically for each actual call context, including recursive calls.

Our approach provides a simple way to adapt the existing applications to context-awareness. Assuming good object-oriented design, adaptation does not require any changes in the legacy applications. This enables the (re-)engineering of self-adaptive and performance-portable (legacy) applications which makes them run efficiently on modern hardware.

The remainder of the paper is structured as follows: Section 2 gives an overview of CAC and sketches the common manual approach of implementing CAC in legacy applications. It discusses the main principles of aspect-oriented context-aware programming and how it could be applied to the adaptation of legacy applications. Section 3 presents key ideas of our implementation approach. Section 4 assesses AOP-based and manual code adaptation approaches in two legacy applications: Sorting and Matrix-Multiplication. More specifically, it assesses the speed-up gained by CAC, the performance overhead of the AOP-based approach, and the lines of code required by a programmer. Section 5 discusses related work, and Section 6 concludes the paper and points out directions of future work.

2 Towards Context-Aware Composition with AOP

In this section, we introduce our AOP-based adaptation approach. After a short introduction to Context-Aware Composition (CAC) in Section 2.1, we discuss the manual steps to adapt to CAC in Section 2.2. Then Section 2.3 defines the common CAC concerns and Section 2.4 demonstrates how to implement these concerns with reusable aspects.

2.1 Context-Aware Composition

CAC is the runtime context dependent binding of a call in one component to a matching callee out of a set of matching callee variants defined by other components. In this paper, we assume that CAC is used to improve application runtime performance (while other optimization goals like decreasing energy or memory consumption are possible, too). Optimization is achieved by binding the caller with the currently best-fit callee variant depending on the actual runtime context. Context-aware composition operates on the following concepts:

Formal context is a call site with formal context parameters. Formal context parameters can include (abstractions and selections of) the formal call parameters (e.g., the problem size), the locally assessable system state, and even the system environment (e.g., the number of available processors).

Actual context is a valuation of the formal context parameters at runtime before a corresponding call.

Component interface is a callee interface that can be bound to the call site of a context. Formally, it is an abstract method with pre-conditions and post-conditions always implied by and always satisfying the call site.

Component variant is a callee implementation, formally, a *co-variant* subtype of the component interface. The implementation variants can range from alternative algorithms and data structures to alternative schedulings or exploitations of hardware accelerators. Note that not all implementation variants need to fit all actual call contexts (co-variance) as long as there is always one variant matching each possible actual call context.

Utility function is an optimization goal function mapping an actual context and a component variant to numerical values representing the variant's performance, footprint, memory consumption, etc. (or combinations thereof) in that context.

Context-aware composition requires a **learning** phase, where the component variants are tested for different actual context and the champion variant of each context is determined. This information is extrapolated and interpolated using machine learning from a total mapping of contexts to champion variants. This mapping function is used in the actual **composition** phase for selecting the presumably optimal variant for each actual context.

While in principle learning and composition may happen offline or online (depending on whether the formal context parameters allow for a static actual context evaluation and binding or not), offline learning and online composition are the most common cases, which are also the basis of the present work.

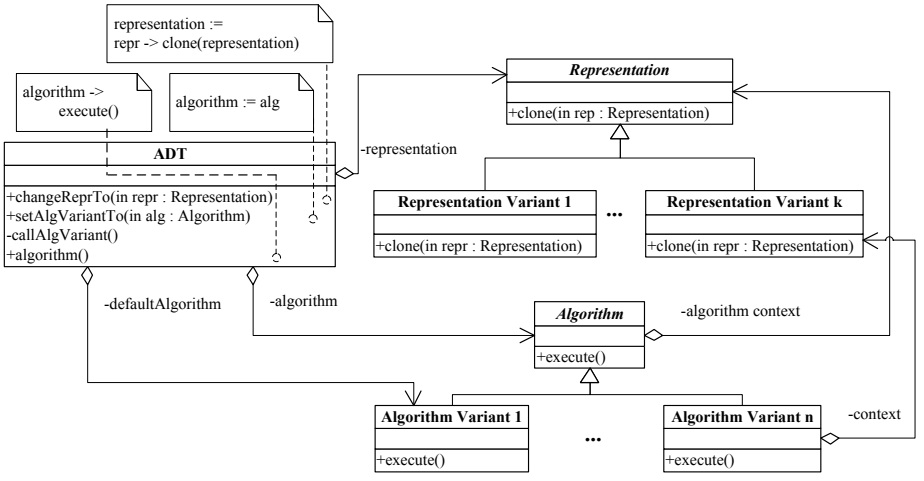


Fig. 1. Object-oriented design for adaptation to CAC

2.2 Prerequisites for Adaptation to Context-Aware Composition

For applying automated adaptation of legacy applications to CAC, we require a good object-oriented design following appropriate design patterns. If an application is not designed accordingly, the design must be established in a manual refactoring/reengineering step.

We need to separate component variants from component interfaces and to distinguish the stateful from the stateless components in such a way that they both could be changed independently. Therefore, we assume the *Strategy* design pattern [6] as depicted in Figure 1: each abstract data type ADT (component) encapsulates states and algorithms operating on that state. It separates the abstract state from its representation and the abstract algorithms from their implementations. All algorithm implementations need access to the state representations (general or special). Besides setting and getting the state or invoking the algorithms, users of the ADT can also control the implementations.

More specifically, `ADT` is configured with concrete Strategy objects determining `Algorithm Variant` and `Representation Variant`. The `ADT` maintains a reference to these objects and defines an interface that lets the algorithm variants uniformly access the data. Algorithm and representation variants are implemented separately by subclasses of the abstract `Representation` and `Algorithm` classes. The `ADT` exposes means to changes the data representation variant (`changeReprTo`) and the algorithm variant (`setAlgVariantTo`). Calls to an abstract algorithm are redirected to the current algorithm variant (`callAlgVarinat`) using the current representation.

Table 1. Application-specific and general CAC concerns

	Application-specific CAC concern	General CAC concern
Design	Formal context, Actual context assessment function.	-
Learning	Sample context and test data generator.	Variant testing, Utility function, Training data generation, Machine learning.
Composition	-	Actual context assessment, Binding to the champion variant.

2.3 Concerns of Context-Aware Composition

Once the prerequisites are established, the following concerns have to be defined: the formal context affecting the choice of the implementation variants, and functions assessing the actual context at each corresponding call site. These concerns cannot be guessed automatically and have to be implemented manually for each ADT. The call sites still bind to the ADT operations on interface level; they neither invoke functions for assessing the actual context nor explicitly control the implementations of representations or algorithms.

For the offline **learning** phase, test data needs to be generated for each sample of the actual contexts. This test data generator needs to be implemented manually for each abstract representation. The learning infrastructure is independent of the specific ADT; it generates test data, assesses the actual context, invokes all implementation variants (all admissible combinations of algorithm and representation implementation), measures performance (or any other utility function) of each combination, and captures the best variant together with the actual context. This training data is an input to a machine learning infrastructure which creates a total mapping from actual context to a champion variants.

The actual **composition** phase at runtime is responsible for dispatching an actual context to its champion implementation variant. The responsible infrastructure is again independent of the actual ADT. It assesses the actual call context, changes representation and algorithm implementation to the presumable champion combination, and dynamically invokes the algorithm. Table 1 summarizes the application-specific and general concerns of the design, and the learning and composition phases, respectively.

2.4 Aspect-Oriented Context-Aware Composition

To separate the profiling/learning from the composition phases and to avoid changing legacy applications (except for refactoring the code to guarantee the prerequisites described in Section 2.2), we use the *Template* design pattern [6]. The pattern implements the common behavior of profiling/learning and composition once and for all applications that have established prerequisites for adaptation to CAC. The common behavior of profiling/learning is implemented in a template method `train` of the `Profiler` class, cf. Figure 2. The *train* method profiles all algorithm variants with different data

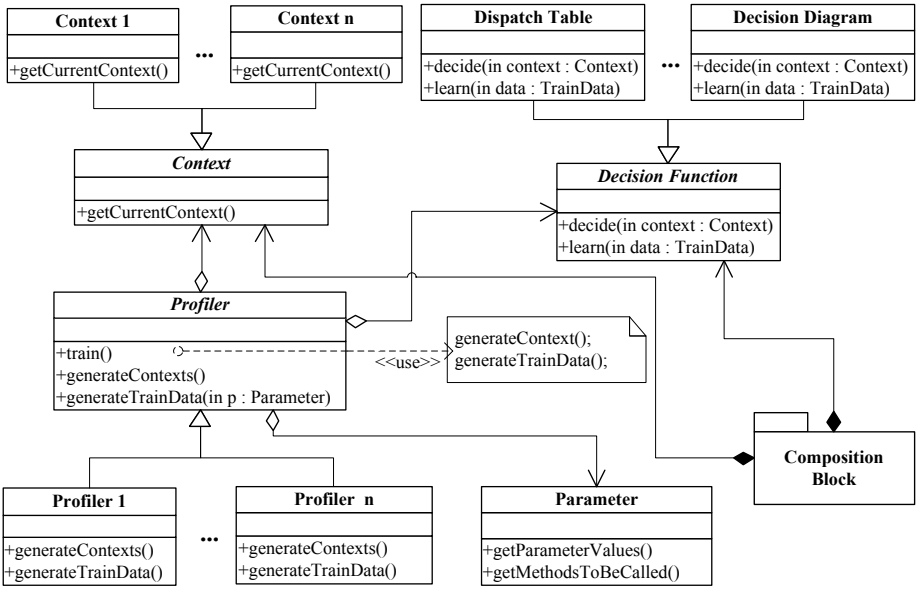


Fig. 2. Design of profiling/learning phase in CAC

representations and captures the best-fit variant. It is defined as a skeleton algorithm deferring the implementation of some steps to subclasses that are specific for concrete applications. These steps are `generateContext` and `generateTrainData`. The former operation samples actual contexts while the second operation generates the input data to the algorithm variants based on these samples. `Context` is an abstract class with an abstract `getCurrentContext` method that provides the actual context; this method implementation is again deferred to application-specific implementations. Finally, the best-fit variant for each actual context is captured and abstracted to a final `Decision Function` using machine learning. The actual learning technology can be exchanged as well, based, e.g., on Dispatch Tables, Decision Diagrams, etc. [7].

The common behavior of composition is separated (cf. the `Composition` block of Figure 2 and Figure 3 for details). Composition only needs access to evaluate the actual context via `getCurrentContext` and to decide on the corresponding best-fit variant via `decide` in the `Decision Function`.

The program points of interposition, where the method calls to alternative algorithm variants occur, are theoretically known. During the learning phase, the method interposition occurs in recursive calls of the algorithm implementation variants. In contrast to that, at runtime, it occurs at every call site to a method with implementation variants. However, regardless of the phase, the call sites, and the target method, the same composition mechanism applies, i.e., the same code fragments have to be inserted at these call sites.

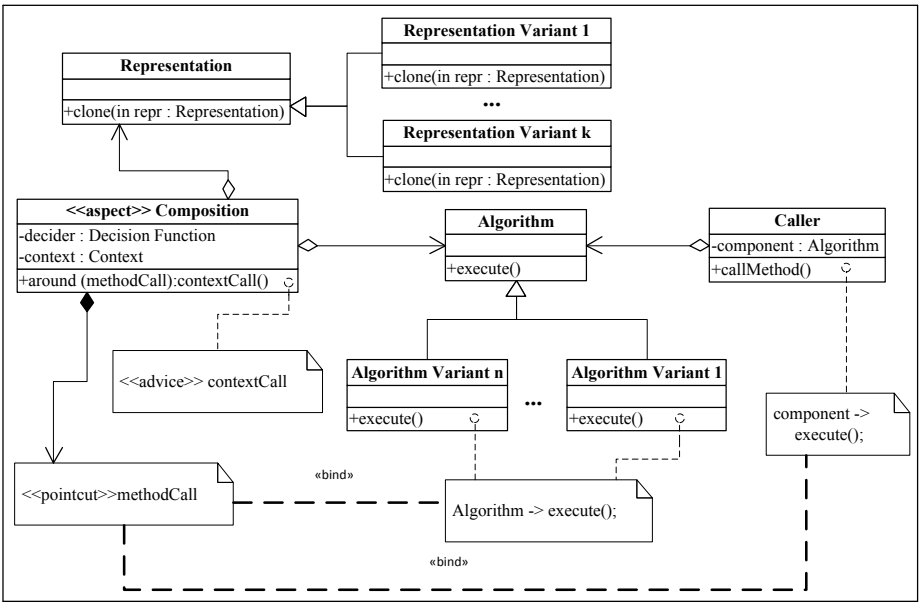


Fig. 3. Design of the Composition Block - AOP-based composition phase in CAC

Because the set of interposition points cut across a number of application module boundaries, they can be defined as cross-cutting concerns. AOP is designed to handle these concerns by providing a mechanism, called *aspect*, for expressing and automatically incorporating them into a program[1] [8]. That is, the points of method interposition are defined once, at one place, making them easy to understand and maintain. Using the terminology of AOP, we refer to an interposition program point as a *join point*, a composition of joint points as a *pointcut*, a set of actions to be executed at a join point as an *advice* and the unit of a program that encapsulates these constructs as an *aspect*.

One of the important properties for adapting existing legacy codes to CAC is that the program modules require no modifications to be advised by the aspects. This is achieved by a process called *weaving* that occurs at build or runtime. It adds the ability to replace method bodies with new implementations, inserts code before, after, or around method calls, and what is most important associates new state and behavior with existing classes. Therefore, expressing context dependencies in aspects follows naturally while separating the definitions of join points, the definition of actual context, and the definition of advice.

Figure 3 complements the profiling/learning phase presented in Figure 2 and shows AOP-based adaptation to CAC. To model aspects in the UML diagram, we used the bottom-up approach presented in [9].

[1] http://www.ibm.com/developerworks/rational/library/2782.html

The aspect `Composition` modeled as a class defines the pointcut `methodCall` and the advice method `contextCall`. The pointcut picks the set of join points where the set of actions defined by the advice method are executed. To actually perform the advice, the aspect requires a concrete `Context` class evaluating the actual context, and a `Decision Function` class to decide the best-fit variant. The pointcut `methodCall` as part of the aspect `Composition` binds the aspect to occurrences of `execute` operations that occur in some `Caller` or even recursively in an `Algorithm Variant`.

3 Implementation Details

In our implementation we used Java/JDK 1.6, as the main programming language both for the AOP adaptation system and for exiting legacy code. For a better understanding of the legacy code adaptation to CAC we consider the example from the sorting problem which is used in our experiments in the next section. In what follows we present a snippet of the actual code of the `QuickSort` algorithm that sorts an array `arr` of general type `E`:

```
public class QuickSort {
(1) public boolean quick_sort(E[] arr, int l, int u) {
(2) ...
(3) quick_sort (arr, l, j+1);
(4) quick_sort (arr, j+1, u);
(5) return true;}}
```

The next snippet depicts manual adaptation of legacy code to CAC that fulfils the required prerequisites regarding object-oriented design discussed in Section 2.2 by implementing the component interface `Sort`.

```
public interface Sort {
(1) public boolean sort(E[] arr, int l, int u);
}
public class QuickSort implements Sort {
(2) public boolean sort(E[] arr, int l, int u) {
(3) ...
(4) sort_dispatch (arr, l, j+1);
(5) sort_dispatch (arr, j+1, u);
(6) return true;}
protected static boolean sort_dispatch (E[] arr, int l, int u) {
 (7) Object [] params = context.getCurrentContext(arr, i, u);
 (8) Sort m = decider.decide(params);
 (9) return m.sort(a, l, u);}}
```

All original calls to an algorithm variant `quick_sort` are manually replaced by a (monomorphic) call to a decider method `sort_dispatch()`, cf. lines 4–5, that based on the current context chooses the best-fit component variant (lines 7–8), followed by a polymorphic call to the instance of the best-fit algorithm variant implementation, cf. line 9.

For the implementation of CAC aspect, cf. Figure 3, we used AspectJ 1.6.12[2]. AspectJ is an AOP environment for the Java language implementing join points, pointcuts, advices, and aspects. At compilation time, the AspectJ compiler inserts the codes defined in an aspect to the existing Java codes and a standard Java compiler compiles these codes to the final class files of the program. For our experiments, we only used algorithm variants while neglecting the data representation variants.

To model algorithm variants and to maximally minimize the change of legacy code, we complement AOP with reflective programming (Java Reflection API's[3]) that allows accessing class definitions at execution time. To specify the methods to be intercepted and the program points where interception has to occur, we exploit Java annotations. We created an `@CACMethod` annotation identifying the methods with alternative algorithm variants.

During training when the algorithm performance for different actual contexts was measured, the alternative method variant objects are obtained by reflection instantiated from algorithm classes containing these methods. Training invokes these methods using `invoke()` of `java.lang.reflect.Method`.

CAC requires the interception of certain method calls to algorithm variants. These calls can be represented by a set of join points that are eventually picked up by the pointcut `methodCall`. It is worth mentioning that not every call to an algorithm variant has to be intercepted when measuring the algorithm performance, only recursive calls selecting a new best-fit variant based on the new actual context (e.g., a smaller problem size). This is achieved by the **call** pointcut that matches all calls within a method with `@CACMethod` annotation.

```
pointcut methodCall(): call(@CACMethod * *(..));
@Around(methodCall())
public Object callContext(ProceedingJoinPoint pjp){
 (1) Object [] params = context.getCurrentContext(pjp.getArgs());
 (2) Method m = context.decider.decide(params);
 (3) return m.invoke(getMethodInstance(m), pjp.getArgs());}
```

Thus, returning to the actual code of `QuickSort` algorithm, the `quick_sort` method will be annotated by `@CACMethod` and the join points will occur at lines 3 and 4. Notice that `QuickSort` in this case has to implement `Sort` interface.

An AspectJ advice can be executed `@Before`, `@After` or `@Around` a certain join point. The advice `contextCall` uses the `@Around` annotation. In our case, it makes sure that the original method does not execute at all. The arguments to this method are extracted from the first parameter of the `callContext` (of type ProceedingJoinPoint) using `getArgs`. Thereafter, the actual context is received (line 1) and used for determining a best-fit method for this context from a `decider` instance, in our case a dispatch table, cf. line 2. The dispatch table was implemented as a simple two-dimensional array capturing the corresponding number of algorithm variants. The aspect knows all alternative method variant objects and class instances to reflectively invoke the best-fit

[2] http://www.eclipse.org/aspectj/
[3] http://www.ibm.com/developerworks/library/j-dyn0603/

variant method during the execution of the program. The value returned by the aspect advice is the return value of the best-fit method execution and is the return value seen by the caller of the initial method.

4 Experiments

For our experiments, we picked two applications: Sorting and Matrix-Multiplication. For both, implementations with the required object-oriented design as well as manual CAC implementations existed already. Algorithm variants use textbook implementations [10] and we did not optimize them further. Especially, parallelization of these algorithms did greedily create new threads when admitted by the essential program dependencies regardless of the actual number of cores available. Sorting comes in the well-known variants of Selection-Sort, QuickSort, MergeSort, and two parallel versions of QuickSort and MergeSort. The latter fork a new thread for one of the two recursive sub-problem calls in each divide step. Matrix-Multiplication implementation variants include the classic algorithm based on three nested loops (referred to as ProductInlined later on), a variant reducing the problem to matrix-vector and then further to vector-vector multiplications, a recursive variant based on eight multiplications of sub-matrixes of one quarter the size of the original, and the famous Strassen algorithm with only seven sub-matrix multiplications. Additionally, the recursive variant was parallelized; it forks new threads for seven of the eight recursive sub-matrix multiplications in each recursion step.

Sorting is an extreme problem for CAC in the sense that the ratio of dynamic variant selection and payload is rather high. Therefore, the performance overhead in this application gives insights on expected upper bounds of this overhead in more common cases like Matrix-Multiplication.

All experiments are executed on two different multi-core machines with native JVMs virtual machine parameters[4]: (M1) a 2 core Fujitsu Siemens Esprimo Mobile v5515 PC running Windows XP (2002, SP 3) on an Intel Dual Core T5300 at 1.73GHz and 1.75GB RAM, and (M2) an 8 core Server Dell Precision WorkStation T7400 running Windows 7 Enterprise 32-bit on an Intel 8 Core Xeon E5410 at 2.33GHz and 8GB RAM(3GB RAM usable).

4.1 Overall Performance

In this section we compare the overall performance of the CAC applications using the AOP-based and manual-based adaptation approaches. In the deployment phase, we constructed dispatch tables for the different multi-core machines (M1, M2). The formal contexts are problem size N and core availability P. For sorting, N was the array size sampled at powers of two between $2^0 \ldots 2^{16}$; for matrix-multiplication, N is the number of rows (columns) of the (square) matrixes sampled as 1, 16 up to 256 with step of 16. For both problems, P is a Boolean encoding whether or not a free core is available.

[4] -Xms40m -Xmx384m.

Table 2. Speed-up of different CAC approaches for Sorting

Platform	Problem size	QuickSort[*msec*]	Aspect	Manual (Aspect)	Manual (Manual)
PC 2 cores (M1)	50,000	34.5	1.46	1.57	2
	500,000	661.1	2	2.1	2.5
	1,000,000	1531.2	2.2	2.3	2.8
Server 8 cores (M2)	50,000	21.7	1.2	1.5	1.7
	500,000	324.3	1.5	1.57	1.9
	1,000,000	775.3	1.6	1.75	2

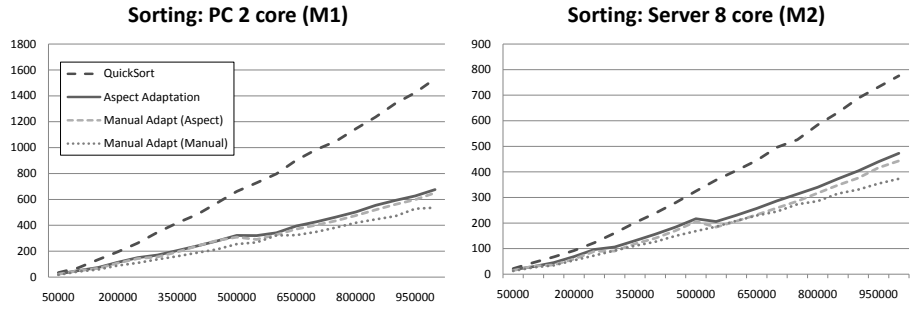
Table 3. Speed-up of different CAC approaches for Matrix-Multiplication

Platform	Problem size	P.Inlined[*msec*]	Aspect	Manual (Aspect)	Manual (Manual)
PC 2 cores (M1)	100	294.9	1.07	1.01	1.3
	500	41119	1.9	1.9	1.9
	800	172820	2.3	2.3	2.5
Server 8 cores (M2)	100	198.8	2	2.07	2.03
	500	29276	2.97	2.98	3.05
	800	124722	3.4	3.5	3.4

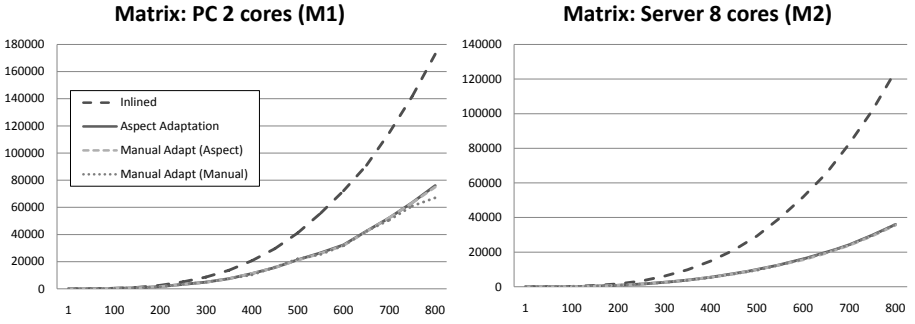
For a fair evaluation, we compared the performance with the manual approach using both the dispatch table produced by aspect-oriented CAC and the dispatch table generated by manual CAC. In the former approach (*Manual Adaptation (Aspect)*), the dispatch table is generated by learning based on the AOP-based CAC approach that also involves reflection. The latter approach (*Manual Adaptation (Manual)*) uses the manually programmed CAC in learning and execution, i.e., it comes without any AOP and reflection. In the pure AOP-based approach *Aspect*, at each invocation of an algorithm variant the advice looks up the best-fit algorithm and executes it using the reflective `invoke` method. The invocation of algorithm variants corresponds to join points in AspectJ. In our experiments the sorting problem has 8 join points that are picked up by the pointcut and the matrix multiplication problem has 35 join points. In both manual approaches, all original calls to an algorithm variant are manually replaced by a (monomorphic) call to a decider method that chooses the best-fit variant followed by a polymorphic call to the instance of the best-fit algorithm variant implementation class.

In the experiments, we compare performance with the fastest context-unaware solutions: the sequential QuickSort and ProductInlined, respectively. The third column of Tables 2 and 3 shows their execution time for two platforms and three selected problem sizes. Columns 4–6 show the speed-up of the CAC approaches relative to the execution time of QuickSort and ProductInlines, respectively, on the same platform and architecture. For instance, in Table 2 *Aspect* gives a speed-up of about 1.46 for M1 and a problem size of 50,000 array elements as it requires only $34.5msec/1.46 = 23.6msec$ of the corresponding QuckSort execution time.

Altogether, for sorting, the pure AOP-based approach gives a speed-up of 1.2 and 2.2 depending on the problem size and the platform. *Manual Adaptation (Aspect)* has a bit



(a) Sequential QuickSort and CAC approaches in Sorting.



(b) Sequential ProductInlined and CAC approaches in Matrix-Multiplication.

Fig. 4. Homogeneous algorithms and CAC using AOP-base and manual approaches. The *x*-axis displays the problem size, the *y*-axis the time in *msec*.

higher speed-up that is between 1.5 and 2.3. *Manual Adaptation (Manual)* provides the highest speed-up, a factor of up to 2.8. Figure 4(a) shows the experimental results from the sorting problem on the two platforms over a wider range of problem sizes. On M1, the AOP-based approach has an average speed-up of 2.05 over sequential QuickSort over array sizes from 50,000 – 1,000,000 (step 50,000). The manual approaches have an average speed-up of 2.17 and 2.5 for aspect and manually constructed dispatch table, respectively. For M2, the difference between AOP-based and manual adaptation is even smaller: average speed-ups of 1.6 versus 1.7 and 1.9, respectively.

For Matrix-Multiplication, Table 3 shows that AOP-based adaptation speeds up the baseline by factors between 1.07 and 3.4 for M1 and M2, respectively. Similar to sorting, the manual approach is slightly faster and speeds up the baseline by factors up to 3.5. Figure 4(b) shows again the performance for a wider range of problem sizes. The average speed-up of the AOP-based approach over matrix sizes of 1–800 rows (step 50) is on average 1.74 and 2.5 for M1 and M2, respectively. However, unlike in sorting, the difference between this AOP-based and manual approaches is minimal: the *Manual Adaptation (Aspect)* has an average speed-up of 1.76 and 2.61, and *Manual Adaptation (Manual)* an average speed-up of 1.9 and 2.6 for M1 and M2, respectively.

4.2 Performance Overhead

The differences in the speed-ups of AOP-based and manual approaches can be explained by the additional overhead due to AOP and reflection used in the AOP-based approach. In order to measure this overhead in a fair way, we compared the CAC execution time using the AOP-based approach with the execution time of the manual approach where both use the same dispatch table. Hence, both approaches select exactly the same variants all the time.

For sorting, the AOP-based approach introduces an average overhead of 5% and 9% over the corresponding *Manual Adaptation (Aspect)* approach for the 2 and 8 core machines, respectively, as it requires 1.05 and 1.09, respectively, of the corresponding *Manual Adaptation (Aspect)* execution time. The overhead is considerably smaller for Matrix-Multiplication; on average about 3% for M1 and 2% for M2.

The overhead of the AOP-based approach is the result of reflection and aspect calls interceptions. For sorting problem, this happens frequently in the recursive algorithms (QuickSort, MergeSort and their parallel implementations). These algorithms are extreme in their ratio between decision points (recursive calls) and workload. This explains the difference in the speed-up of the AOP-based and the manual approaches. In contrast, Matrix-Multiplication shows a rather low overhead. In the recursive algorithms, the ratio between decision points and workload is rather low as can be expected from many other applications.

In both example applications, the overhead of CAC is more than compensated by the speed-up compared to the homogenous variants.

4.3 Lines of Code

The slightly higher performance of the manual CAC approaches are paid by a higher programming effort. In order to measure the effort required to adapt a given legacy application to CAC, we used the lines of code (LOC) metrics. It measures the number of lines of a program's source code. Specifically, we measured LOC required for the manual CAC adaptation, referred to as LOC_M, and LOC required for the AOP-based adaptation, referred to as LOC_A. The manual adaptation may reuse some parts of the existing legacy application and change only some lines of the code. LOC_M counts only lines that require additional changes or have to be added to achieve CAC. The AOP-based approach reuses the code implementing general CAC concerns. LOC_A counts only the application-specific code, cf. Table 1. Finally, the programming effort improvement due to the AOP-based approach is simply assessed with a metrics $PI = LOC_M/LOC_A$. For the sorting problem, $LOC_M = 154$ and $LOC_A = 115$ lead to quite a productivity improvement of $PI = 1.4$. For Matrix-Multiplication, $LOC_M = 383$ and $LOC_A = 134$ lead to $PI = 2.8$.

The code changed in the manual adaptation is actually spread throughout the legacy application. It requires some time and effort to identify which lines have to be changed for CAC adaptation, which is not assessed by our metrics PI. For instance the total lines of code (LOC_T) of the sorting problem is 441. Thus, the tangling ratio $TR = LOC_M/LOC_T$ is 0.35. For the matrix multiplication problem $LOC_T = 1500$. Thus, $TR = 0.25$. Therefore, the actual improvement of the programming (and later main-

tenance) effort is even higher due to the encapsulation of the respective concerns in AOP.

Altogether, the experiments showed that our approach can effectively adapt existing legacy codes to CAC which makes them run efficiently even on multi-core machines. Even though the speed-ups of applications with CAC adapted manually is slightly higher than with the AOP-based approach, the performance overhead is quite small. Moreover, AOP-based adaptation requires a smaller programming effort and the resulting systems still significantly outperform the fastest heterogeneous applications. Based on these observations, we claim that the AOP-based approach can be widely used by software developers to adapt legacy applications to CAC.

5 Related Work

In general, context-aware computing allows application changes depending on the location of the user, hosts, accessible devices, etc [11]. Variants differ in the applied problem domain, which can include, e.g., mobile, Web, e-services [12,13,14], and composition technology. Context-awareness can be achieved with Context-Oriented Programming [15,1,2]. Context-aware composition aims at dynamically optimizing applications in changing call contexts and available resources in the system environment [16,17,18].

The optimization of domain-specific libraries for linear algebra or signal processing is a natural target for optimized composition, called autotuning. Because the domain and code base is limited and statically known, computations can often be described in a restricted domain-specific language from which variants can be generated and tuned automatically. Well-known examples include the library generators ATLAS [19] for basic linear algebra computations, and FFTW [20] and SPIRAL [21,22] for transformations in signal processing. Even though these approaches are similar to CAC, they do not have any separated concerns for introducing autotuning to any other domain than the one they were defined for. Additionally, the dynamic program optimization can be achieved by the tracing just-in-time compilers that determine frequently executed traces (hot path and loops) in running programs and focus their optimization effort by emitting optimized machine code specialized to these traces [23]. This strategy is beneficial for dynamic languages [24], and has been recently implemented for Microsoft Common Intermediate Language (CIL) [25] and Java [26,27].

Optimized composition has been proposed as an optimization technique also in the more general context of component based systems, where the programmer is responsible for annotating components so that their composition can be optimized for performance. However, only few approaches consider recursive components with *deep composition*, and only few consider the co-optimization of the selection of implementation variants with other variation possibilities, such as the layout and data structure of operands or scheduling [28,29,30,31]. For instance, Andersson *et al.* use CAC to compose and optimize implementation variants of data structures and recursive algorithms, considering Matrix-Multiplication as a case study [16]. Kessler and Löwe [17,18] consider optimized composition at the level of annotated user-defined components (i.e., not limited to closed libraries) together with scheduling, resource allocation and other optimizations, which allows for simultaneous optimization. They use Sorting as one of their

case studies. Yu and Rauchwerger [32] investigated dynamic algorithm selection for reductions in the STAPL [33] library for sorting and matrix computations. Olszewski and Voss [34] proposed a dynamic adaptive algorithm selection framework for devide- and -conquer sorting algorithms in a fork-join parallel setup. They use dynamic programming algorithm to select the best sequential algorithm for different problem sized. A detailed discussions on these approaches can be found in our previous work [7], where we compare the accuracy and the performance of different machine learning approaches in CAC. However, regardless of all the benefits of CAC, it still requires a considerable effort to design and implement applications accordingly. This becomes even more complicated when adapting existing complex legacy applications to CAC. Therefore, the current work presented in this paper is orthogonal to the previous work as is suggests a general way for *introducing* CAC to legacy applications.

The notion of a context-aware aspect with behavior depending on the context was first presented by Tanter *et al.* [5]. They analyze the appropriateness of the support of AOP languages for expressing aspect for accessing the information associated to the current application contexts. Moreover, the authors propose an open framework for context-aware aspects that supports the definition of context-awareness constructs for aspect. It also includes the ability to refer to past contexts, and to provide domain and application specific contexts. CAC applications additionally require *learning* from past context experiences about the best-fit component variants to be executed in an actual context.

There are ongoing efforts supporting the development of context-aware computing systems. For instance, David *et al.* [3] present the WildCAT system which is a general context awareness toolkit. It provides a way for Java developers to make their software context-aware, by providing APIs (and shared low level code) to maintain various events (contexts) occurring at program runtime. Using this system in our AOP-based approach could avoid application-specific context implementation. Delicato *et al.* [13] propose a framework for developing context-aware applications for mobile computing. The framework is aspect-oriented and is implemented in AspectJ. It provides a set of default adaptive concerns common to mobile applications along with concrete aspects implementing these concerns. However, besides implementing a GUI interface, the developer has to specify which concerns have to be used at which program points that are specific to the application to be implemented. Li *et al.* [35] present an AOP-based approach to address context-aware Web service composition. Their approach semantically composes different Web services whenever the context changes. This work shows that context weaving is suitable for the implementation of CAC services when the context is dynamic and hard to predict. Although these approaches allow for implementing new CAC systems, they still remain domain specific and, in contrast to our work, do not consider the adaptation of existing legacy application to CAC for improving application performance.

6 Conclusions and Future Work

Context-aware composition allows improving the performance of applications on modern multi-core hardware by separating the concerns of design, deployment, and execution of component adaptations. It provides reusable, performance-portable systems

selecting (sequential and parallel) component variants for actual (call and hardware) contexts. Some of these concerns are general regardless of the application domain.

However, manually introducing CAC in legacy applications can be time-consuming since it requires additional effort for changing application design and adapting the existing code. For automating this process, we proposed an AOP-based adaptation approach. Due to the strong encapsulation of AOP programs, the general CAC concerns can be reused easily for different applications and may also be used in the development of new CAC applications from scratch; developers can focus on their own application design and add the CAC aspect later rather than implement the CAC design pattern manually. Our main objectives were to impose as little execution overhead as possible and to require as little as possible changes from the (legacy or core) application.

Our AOP-based approach was evaluated on Sorting and Matrix-Multiplication applications in terms of performance and programming effort relative to the manual adaptation approach. The experiments showed that applications derived by our AOP-based adaptation speeds up the execution time by factors of up to 2.2 and 3.4 for Sorting and Matrix-Multiplication, respectively, on multi-core machines with two and eight cores. The application manually adapted and optimized to CAC achieved a slightly higher speed-up (2-9% higher). However, development and maintenance effort for the manual adaptation is higher than for the AOP-based approach. Manual adaptation requires almost three times the code of the AOP-based approach and is additionally spread throughout the application and not nicely encapsulated as in the AOP-based adaptation. These results suggest that an AOP-based adaptation to CAC can effectively improve the performance of existing applications with a moderate transition effort.

As a matter of future work, we will handle data representation variants and combine their selection with the best-fit algorithm variants at a runtime using AOP. Also, the aspect in our current implementation can only adapt explicit method calls to alternative variants. However, legacy application might also contain more indirect and dynamic calling mechanisms including dynamic invocations, stub-invocations etc. Therefore, we have to generalize the current aspect implementation to include these possible pointcut types. Future work also includes experimenting with other legacy applications and reevaluating our approach in online learning scenarios as required in self-adaptive systems; this should provide a more general approach to CAC code adaptation. Finally, we need to evaluate our approach in a real industrial project and measure gained speed-ups and programming effort required for CAC adaptation.

Acknowledgments. Project ”Context-Aware Composition of Parallel Components” (2011-6185). Partially funded by the Swedish Research Council (Vetenskapsrådet).

References

1. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Obj. Tech. 7(3), 125–151 (2008)
2. Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming: an overview of contextl. In: Proc. of the 2005 Symp. on Dynamic Lang. DLS 2005, pp. 1–10. ACM, New York (2005)

3. David, P.-C., Ledoux, T.: WildCAT: a generic framework for context-aware applications. In: Proc. of the 3rd Int. Workshop on Middleware for Pervasive and Ad-hoc Computing. MPAC 2005, pp. 1–7. ACM, NY (2005)
4. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C.: Composing adaptive software. Computer 37, 56–64 (2004)
5. Tanter, É., Gybels, K., Denker, M., Bergel, A.: Context-Aware Aspects. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089, pp. 227–242. Springer, Heidelberg (2006)
6. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
7. Danylenko, A., Kessler, C., Löwe, W.: Comparing Machine Learning Approaches for Context-Aware Composition. In: Apel, S., Jackson, E. (eds.) SC 2011. LNCS, vol. 6708, pp. 18–33. Springer, Heidelberg (2011)
8. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
9. Kandé, M.M., Kienzle, J., Strohmeier, A.: From AOP to UML—A bottom-up approach. In: Workshop on Aspect-Oriented Modeling with UML (March 2002)
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, New York (2001)
11. Schilit, B., Adams, N., Want, R.: Context-aware computing applications. In: Proc. of the 1994 First Workshop on Mobile Comp. Systems and Applications, pp. 85–90. IEEE Computer Society, Washington, DC (1994)
12. Lyons, K., Want, R., Munday, D., He, J., Sud, S., Rosario, B., Pering, T.: Context-aware composition. In: HotMobile (2009)
13. Delicato, F.C., Santos, I.L.A., Pires, P.F., Oliveira, A.L.S., Batista, T., Pìrmez, L.: Using aspects and dynamic composition to provide context-aware adaptation for mobile applications. In: Proc. of the 2009 ACM Symp. on Applied Computing. SAC 2009, pp. 456–460. ACM, New York (2009)
14. Baresi, L., Bianchini, D., Antonellis, V.D., Fugini, M.G., Pernici, B., Plebani, P.: Context-aware composition of e-services. In: Proc. of VLDB Workshop on Technologies for E-Services, pp. 7–8 (2003)
15. von Löwis, M., Denker, M., Nierstrasz, O.: Context-oriented programming: beyond layers. In: Proc. of the 2007 Int. Conf. on Dynamic Lang.: in Conjunction with the 15th Int. Smalltalk Joint Conf. 2007, pp. 143–156. ACM, New York (2007)
16. Andersson, J., Ericsson, M., Kessler, C.W., Löwe, W.: Profile-Guided Composition. In: Pautasso, C., Tanter, É. (eds.) SC 2008. LNCS, vol. 4954, pp. 157–164. Springer, Heidelberg (2008)
17. Kessler, C., Löwe, W.: A framework for performance-aware composition of explicitly parallel components. In: PARCO, pp. 227–234 (2007)
18. Kessler, C., Löwe, W.: Optimized composition of performance-aware parallel components. Concurrency and Computation: Practice and Experience (2011)
19. Whaley, C., Petitet, A., Dongarra, J.J.: Automated empirical optimization of software and the atlas project. Parallel Computing 27, 2001 (2000)
20. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proceedings of the IEEE 93(2), 216–231 (2005)
21. Moura, J.M.F., Johnson, J., Johnson, R.W., Padua, D., Prasanna, V.K., Püschel, M., Singer, B., Veloso, M., Xiong, J.: Generating platform-adapted DSP libraries using SPIRAL. In: High Performance Embedded Computing, HPEC (2001)
22. Moura, J.M.F., Johnson, J., Johnson, R.W., Padua, D., Prasanna, V.K., Püschel, M., Veloso, M.: SPIRAL: Automatic implementation of signal processing algorithms. In: High Performance Embedded Computing, HPEC (2000)

23. Bala, V., Duesterwald, E., Banerjia, S.: Dynamo: a transparent dynamic optimization system. SIGPLAN Not. 35(5), 1–12 (2000)
24. Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M.R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E.W., Reitmaier, R., Bebenita, M., Chang, M., Franz, M.: Trace-based just-in-time type specialization for dynamic languages. SIGPLAN Not. 44, 465–478 (2009)
25. Bebenita, M., Brandner, F., Fähndrich, M., Logozzo, F., Schulte, W., Tillmann, N., Venter, H.: Spur: a trace-based jit compiler for cil. In: OOPSLA, pp. 708–725 (2010)
26. Zaleski, M., Brown, A.D., Stoodley, K.: Yeti: a gradually extensible trace interpreter. In: Proc. of the 3rd Int. Conference on Virtual Execution Environments. VEE 2007, pp. 83–93. ACM, New York (2007)
27. Gal, A., Probst, C.W., Franz, M.: HotpathVM: an effective jit compiler for resource-constrained devices. In: Proc. of the 2nd Int. Conf. on Virtual Execution Environments. VEE 2006, pp. 144–153. ACM, New York (2006)
28. Brewer, E.A.: High-level optimization via automated statistical modeling. In: PPoPP 1995 (1995)
29. Ansel, J., Chan, C.P., Wong, Y.L., Olszewski, M., Zhao, Q., Edelman, A., Amarasinghe, S.P.: PetaBricks: a language and compiler for algorithmic choice. In: Proc. ACM SIGPLAN Conf. on Progr. Language Design and Implem., pp. 38–49. ACM (2009)
30. Wernsing, J.R., Stitt, G.: Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing. In: Proc. ACM Conf. on Lang., Compilers, and Tools for Embedded Systems (LCTES 2010), pp. 115–124. ACM (2010)
31. Li, X., Garzarán, M.J., Padua, D.: A dynamically tuned sorting library. In: Proc. CGO 2004, pp. 111–124 (2004)
32. Yu, H., Rauchwerger, L.: An adaptive algorithm selection framework for reduction parallelization. IEEE Trans. Par. Distr. Syst. 17, 1084–1096 (2006)
33. Thomas, N., Tanase, G., Tkachyshyn, O., Perdue, J., Amato, N.M., Rauchwerger, L.: A framework for adaptive algorithm selection in STAPL. In: Proc. ACM SIGPLAN Symp. on Princ. and Pract. of Parallel Programming, pp. 277–288. ACM (June 2005)
34. Olszewski, M., Voss, M.: An install-time system for the automatic generation of optimized parallel sorting algorithms. In: Proc. PDPTA 2004, vol. 1 (June 2004)
35. Li, L., Liu, D., Bouguettaya, A.: Semantic based aspect-oriented programming for context-aware web service composition. Inf. Syst. 36, 551–564 (2011)

Challenges for Refinement and Composition of Instrumentations: Position Paper

Danilo Ansaloni[1], Walter Binder[1], Christoph Bockisch[2], Eric Bodden[3], Kardelen Hatun[2], Lukáš Marek[4], Zhengwei Qi[5], Aibek Sarimbekov[1], Andreas Sewe[3], Petr Tůma[4], and Yudi Zheng[5]

[1] University of Lugano, Switzerland
{danilo.ansaloni,walter.binder,aibek.sarimbekov}@usi.ch
[2] University of Twente, The Netherlands
c.m.bockisch@cs.utwente.nl, k.hatun@ewi.utwente.nl
[3] Technische Universität Darmstadt, Germany
eric.bodden@ec-spride.de, andreas.sewe@cased.de
[4] Charles University, Czech Republic
{lukas.marek,petr.tuma}@d3s.mff.cuni.cz
[5] Shanghai Jiao Tong University, China
{qizhwei,zheng.yudi}@sjtu.edu.cn

Abstract. Instrumentation techniques are widely used for implementing dynamic program analysis tools like profilers or debuggers. While there are many toolkits and frameworks to support the development of such low-level instrumentations, there is little support for the refinement or composition of instrumentations. A common practice is thus to copy and paste from existing instrumentation code. This, of course, violates well-established software engineering principles, results in code duplication, and hinders maintenance. In this position paper we identify two challenges regarding the refinement and composition of instrumentations and illustrate them with a running example.

Keywords: Instrumentation, composition, aspect-oriented programming, domain-specific languages.

1 Introduction

Many dynamic program analyses, including tools for profiling, debugging, testing, program comprehension, and reverse engineering, rely on code instrumentation. Such tools are usually implemented with toolkits that allow for the careful low-level optimization of the inserted code. While this low-level view is needed to keep the overhead incurred by dynamic program analyses low, it currently brings with it a lack of support for refining and composing the resulting instrumentations. Code duplication, caused either by copy and paste or by the reimplementation of common instrumentation tasks, is therefore a common code smell of many instrumentation-based tools; it is known to be error-prone and to hinder software

T. Gschwind et al. (Eds.): SC 2012, LNCS 7306, pp. 86–96, 2012.

maintenance. This is all the more problematic, as errors in low-level code are notoriously hard to find.

Before we discuss the challenges arising from the refinement and composition of instrumentations, we first define our terminology and context. An *instrumentation* selects certain sites in the code of a given base program—so-called *instrumentation sites*—and inserts code to be executed whenever the control flow reaches these sites. The *inserted code* must not change the semantics of the base program; it must complete after a finite number of instructions without throwing any exception into the base program and may read but not write any memory location accessed by the base program. Inserted code must thus resort to dedicated memory locations to pass data between different instrumentation sites. For example, local variables invisible to the base program may be used to pass data between several instrumentation sites within the same method body. Likewise, thread-local variables or global variables may be used to pass data between instrumentation sites in different methods. The inserted code typically invokes analysis methods, e.g., to update a profile. We call the classes defining those methods the *runtime classes* of the analysis.

Suitable refinement and composition mechanisms for instrumentations need to address the following two general challenges:

1. *Specification and enforcement of constraints*: Instrumentations usually fail to state important assumptions that are crucial for the instrumentations' correctness in general and when refining or composing instrumentations in particular. Such assumptions may constrain the following.
   (a) *Instrumentation sites*: The selection of sites by the different instrumentations must be consistent; different instrumentations, e.g., may need to refer to the same part of a program, regardless of whether they share common instrumentation sites or not.
   (b) *Instrumentation ordering*: Composing instrumentations often requires defining ordering constraints, not only for the instrumentations as a whole but possibly even for each instrumentation site that they target.
   (c) *Data passing*: Instrumentations may declare variables to pass data between instrumentation sites. Each of these variables has to be initialized by one instrumentation before it can be read by another.
2. *Avoiding hard-coded dependencies*: Usually, the inserted code has hard-coded dependencies on specific runtime classes. Such dependencies typically resemble invocations of static methods or constructors of runtime classes in the inserted code. When refining or composing instrumentations, these dependencies may need to be changed to use different runtime classes.

As original contribution, in this position paper we study the aforementioned challenges regarding refinement and composition of instrumentations and illustrate them with a running example (Section 2). Section 3 discusses related work and Section 4 concludes.

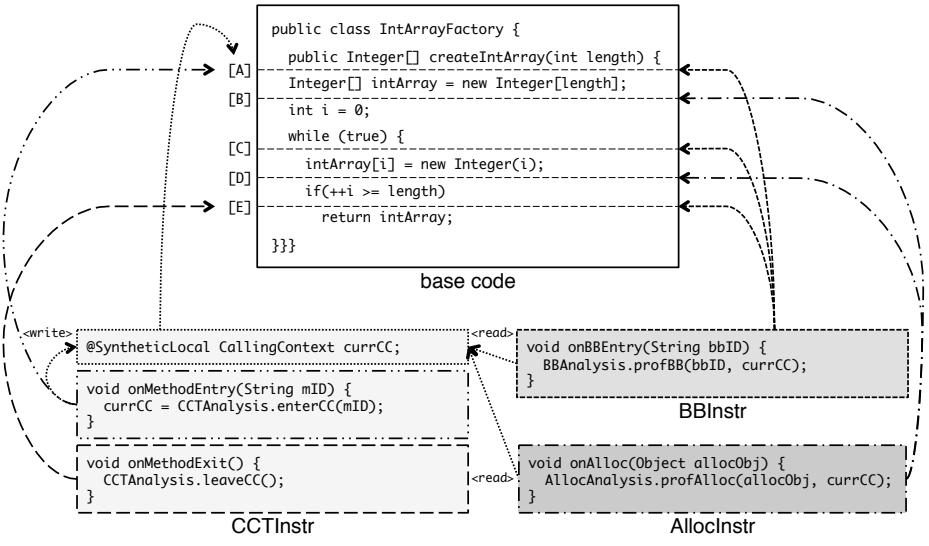


Fig. 1. Instrumentation sites for a composition of three instrumentations: calling context profiling (*CCTInstr*), basic block profiling (*BBInstr*), and object allocation profiling (*AllocInstr*)

2 Challenges

There are several challenges that make refinement and composition of instrumentations difficult. To explain these challenges, we first introduce a running example in which three common instrumentations are composed. Next, we motivate the need for specifying and enforcing instrumentation constraints. Finally, we consider the problem of hard-coded dependencies from inserted code to specific runtime classes.

2.1 Motivating Example

Figure 1 illustrates our motivating example: a composition of three instrumentations (pseudo-code) applied to some base program. While the inserted code is intentionally kept simple, the three instrumentations have interactions that resemble those of complex, real-world analyses.

CCTInstr. The *CCTInstr* analysis maintains a Calling Context Tree (CCT) [1], i.e., a data structure that can be used to store dynamic metrics separately for each individual calling context. To efficiently expose a reference to the CCT representation of the current calling context to the code inserted into the same base-program method by the other two instrumentations, *CCTInstr* declares the synthetic local variable *currCC* that will be mapped to a local variable in each instrumented method. *CCTInstr.onMethodEntry(...)* represents the code inserted

at instrumentation site `[A]`. We assume the instrumentation framework provides some context information; the string *mID* identifies the base-program method to be instrumented. The runtime class *CCTAnalysis* keeps track of the current calling context for each thread and maintains the CCT. *CCTAnalysis.enterCC(...)* updates the current thread's calling context on method entry and returns a reference to it, which is then stored in *currCC*. *CCTAnalysis.leaveCC()*, inserted at instrumentation site `[E]`, updates the current thread's calling context on method completion.

BBInstr. The *BBInstr* analysis counts how often each basic blocks of code is executed. The code of *BBInstr.onBBEntry(...)* is inserted at the instrumentation sites `[A]`, `[C]`, and `[E]`. As context information provided by the instrumentation framework, *bbID* identifies the executed basic block. *BBAnalysis.profBB(...)* updates a counter corresponding to *bbID* in the current calling context (*currCC*).

AllocInstr. The *AllocInstr* analysis profiles object allocations. The code of *AllocInstr.onAlloc(...)* is inserted at the instrumentation sites `[B]` and `[D]`. As context information, *allocObj* refers to the allocated object. *AllocAnalysis.profAlloc(...)* updates an object allocation counter in the current calling context (*currCC*).

2.2 Specification and Enforcement of Constraints

We now describe the implicit constraints that must be respected to preserve correctness when refining or composing the instrumentations illustrated in Fig. 1.

Instrumentation Sites. To ensure soundness of the CCT, *CCTInstr* must be comprehensively applied to all classes. In contrast, restricting the scope of *BBInstr* and *AllocInstr* does not impair the correctness of the (subset of) collected data. To reduce the runtime overhead of the analysis, it may even be desirable to restrict expensive instrumentations like *BBInstr* to a subset of the base-program classes. To ensure the consistency of one's measurements, however, it is likewise desirable to ensure that *BBInstr* and *AllocInstr* are applied to the same selection of classes.

Instrumentation Ordering. A hard constraint of the instrumentations illustrated in Fig. 1 concerns the synthetic local variable *currCC* used to share the current calling context among the instrumentations. Even if not explicitly stated, both *BBInstr.onBBEntry(...)* and *AllocInstr.onAlloc(...)* expect this variable to be initialized by *CCTInstr.onMethodEntry(...)*. That is, if multiple instrumentations insert code on method entry, *CCTInstr.onMethodEntry(...)* must be applied first. As a consequence, at instrumentation site `[A]`, *CCTInstr.onMethodEntry(...)* has to be inserted before *BBInstr.onBBEntry(...)*. For the other instrumentation sites, no particular ordering is required.[1] Unfortunately, ordering constraints are often

[1] Intuitively, at instrumentation site `[E]`, *CCTInstr.onMethodExit()* should be inserted after *BBInstr.onBBEntry(...)*. But as *CCTInstr.onMethodExit()* does not modify the synthetic local variable *currCC*, the insertion order at `[E]` does not matter.

implicit in the implementation of complex instrumentations. Therefore, composing instrumentations usually requires in-depth knowledge of their implementation to avoid violating any implicit ordering constraint.

Data Passing. Efficient communication between different composed instrumentations is often necessary to reduce runtime overhead. However, the declaration of the name and the type of variables used for such communication is usually hard-coded in the instrumentations; thus, it is difficult to refine them without in-depth knowledge of all implementation details. In our example, both *BBInstr* and *AllocInstr* expect a variable of type *CallingContext* named *currCC*. This constraint hinders composition of instrumentations, as it is often necessary to update the code of some instrumentations to communicate through the same variables.

2.3 Hard-Coded Dependencies

Frequently, instrumentations use static method calls to access runtime classes, often in a desire to avoid the runtime overhead associated with virtual methods. For example, the instrumentations illustrated in Fig. 1 include static calls to methods in the runtime classes `CCTAnalysis`, `BBAnalysis`, and `AllocAnalysis`. These dependencies constrain refinement of runtime classes, as static methods cannot be overridden. Refactoring runtime classes to adhere to the singleton pattern helps mitigate the problem, but the static method returning the singleton instance cannot be refined so as to return an instance of a refined runtime class. To address this issue, mechanisms for dependency injection are needed.

3 Related Work

In the past, both low-level frameworks and aspect-oriented approaches have been used for various instrumentation tasks. While the former are typically more expressive and lead to faster code, the latter may offer more powerful refinement and composition mechanisms. In the following text, we compare the properties of both kinds of approaches. For brevity, we limit the discussion to solutions for the Java Virtual Machine, as it is a widely used deployment platform and has been targeted by a large body of related work.

3.1 Instrumentation Frameworks

Bytecode Manipulation Frameworks. Low level bytecode manipulation frameworks like ASM[2], BCEL[3], Javassist [5], or Soot [15] support the direct generation or transformation of arbitrary bytecode. While they offer the maximally possible control over bytecode instrumentation, composition of instrumentations is not directly supported. When instrumentations are to be developed separately,

[2] See `http://asm.ow2.org/`
[3] See `http://commons.apache.org/bcel/`

they can thus only be composed by applying them sequentially. In this case, however, each instrumentation receives the bytecode resulting from the previous instrumentations as input. Thus, later instrumentations generally cannot distinguish between the original code and code inserted by earlier instrumentations.[4] As a consequence, controlling the composition of instrumentations becomes infeasible. This is also true of the Scala library Mnemonics [11] which is slightly less low-level than the aforementioned frameworks; by exploiting Scala's type system, it ensures that only certain well-formed, type-safe bytecode sequences can be generated.

RoadRunner. Flanagan and Freund [6] propose a framework for composing different small and simple analyses for concurrent programs. Each analysis can stand on its own, but by composing them one can obtain more complex ones: each dynamic analysis is essentially a filter over event streams, and filters can be chained. Per program run, only one chain of analyses can be specified. Thus, it is generally not possible to combine arbitrary analyses; for example, two analyses that filter (e.g., suppress) events in an incompatible way cannot be combined.

DiSL. DiSL [9,18] is a domain-specific language for instrumentation. While it offers high-level abstractions to ease the development of instrumentations, it also gives the programmer fine-grained control over the inserted code. However, DiSL lacks refinement and composition mechanisms.

3.2 Aspect-Oriented Approaches

Aspect-oriented programming (AOP) is frequently used as a high-level, language-based approach to implementing analyses. An analysis roughly maps to one or more aspects; then the instrumentations of the analysis correspond to pairs of pointcuts and advice. Pointcuts select the instrumentation sites and advice define the inserted code, in this analogy. But as aspect-oriented languages typically focus on high-level interaction with the program execution, the available instrumentation sites and context accessible in inserted code is limited. Nevertheless, the requirements for the re-use and composition of aspects are similar to those of analyses and their instrumentations. In the following text, we thus briefly discuss selected aspect-oriented languages and their features with respect to (1) constraining instrumentation sites (*scope*), to (2) specifying the *order* of aspects at shared instrumentation sites, and to (3) *sharing* structure and implementation between aspects (e.g., to realize data sharing).

AspectJ. In the past, the AspectJ language [8] (or derivatives) has been used for implementing dynamic analyses; often alternative compilers are used for this purpose, such as MAJOR [4,16,17] and MAJOR2 [10,13] which allows instrumenting the Java class library unlike the standard AspectJ weaver. In AspectJ,

[4] Soot provides a way to tag statements and bytecode instructions. However, there are no guidelines that would govern or enforce a principled use of this mechanism.

aspects extend classes with pointcuts and advice.[5] Pointcuts are boolean expressions whose operands are again (possibly primitive) pointcuts. They can also be named and then referenced from multiple other pointcuts.

Scope: In an abstract aspect, pointcut expressions may also refer to *abstract* named pointcuts; e.g., the pointcut expression *scope() && call(*.new(..))* selects all constructor call sites at which also the pointcut *scope()* matches, which can be declared as an abstract pointcut. Abstract aspects can be extended whereby concrete expressions must be provided for abstract pointcuts. In this way, an analysis implemented in an aspect can be re-used while the scope for applying the analysis may be reduced, e.g., by specifying an expression for the *scope()* pointcut that only matches within a certain package.

Order: Ordering constraints between instrumentations (i.e., pointcut-advice pairs) can be imposed by explicitly declaring the precedence of entire aspects, possibly external to the aspects in question.

Sharing: Extending an abstract aspect is the only means to code re-use supported by AspectJ. The sub-aspect has to concretize abstract pointcuts and can override virtual methods.

CaesarJ. With respect to the pointcut-advice mechanism, aspects in CaesarJ [2] are very similar to those of AspectJ. Two extensions are relevant for the scoping and sharing issue of this paper, however.

Scope: Additionally, it allows to programmatically deploy and undeploy aspects and to limit their activation to certain threads or objects, thereby refining the scope of an aspect.

Order: CaesarJ provides the same mechanism for declaring aspect precedence as AspectJ.

Sharing: The CaesarJ language extends the Java type system with dependent types, i.e., types which are properties of (aspect) instances. Thus, expressions like *this* can be used in type declarations and the compiler can verify that covariant types are used together consistently.

JAsCo. The JAsCo language [12] extends Java beans with so-called hooks to aspect beans. Similar to AspectJ and CaesarJ, it offers a pointcut-advice mechanism, however, in JAsCo it is composed of several individual concepts which improve re-usability and configurability.

Scope: A hook is similar to an inner class which defines a context-independent pointcut in its constructor. Being context-independent, the pointcut expression does not refer to actual methods but rather refers to the constructor's parameters, which are made concrete upon hook instantiation. Besides its constructor and advice, a hook can contain the possibly abstract method *isApplicable*, which furthermore refines the hook's scope. Hooks are instantiated and deployed using so-called connectors which supply concrete method patterns to the hooks' constructors and implement any abstract *isApplicable* methods.

[5] The extension with inter-type declarations is out of the scope for this paper.

Order: Connectors not only specify the hooks' scope, they also fix their order either explicitly or implicitly through programmatic combination strategies. The latter can also be used to conditionally remove applicable hooks to mimic overriding.
Sharing: JAsCo allows to override hook methods on a per-advised-object basis. In this way aspect beans can be re-used and extended on different contexts.

HyperJ. The HyperJ [14] approach attempts to decompose a program along different dimensions. For each dimension, a partial program, called a hyperslice, is written which must be declaratively complete; functionality not provided but required by a hyperslice must be declared as abstract methods. A control file then governs the composition of hyperslices, which configures how the methods of the hyperslices are matched and merged.

Scope: One matching strategy is to match methods with the same names, but it is possible to compensate mismatches which especially occur when hyperslices are developed independently.
Order: The merging strategy is similarly configurable; if all except one of the definitions are abstract, the merging is trivial. For more than one concrete, matching unit, HyperJ provides merging strategies such as overriding or aggregating the result of the separate unit.
Sharing: In HyperJ, abstract methods, together with appropriate matching and merging strategies, can be used to share functionality among hyperslices.

Composition Filters. The Composition Filters Model [3] is based on the concepts of filters which are applied to method invocations.

Scope: A filter selects invocations based on the method's name and signature and it can perform additional actions or influence the execution of the target method. Filtermodules group filters and can declare data fields holding shared values. They can declare parameters to be used, e.g., for the type of fields or in the filters' expressions for selecting method invocations. A superimposition block has to be declared, possibly in a separate module, which deploys filtermodules on a set of types and provides concrete values for the parameters.
Order: For jointly superimposed filtermodules, a partial order can be specified. Other relations like overriding between filters can be declared in a similar way.
Sharing: A filtermodule can be superimposed multiple times on different type sets and with different parameter values.

Framed Aspects. In the Frames approach, so-called tags may be inserted into the code. For these tags a configuration file can then provide an application-specific replacement.

Scope: The Framed Aspects approach [7] allows the insertion of tags into aspects. Tags can be used, e.g., in place of type or method names, expressions or in patterns used by pointcuts. Thus, the scope of aspects can be refined in the configuration file.

Order: The Framed Aspects approach is independent of a concrete aspect language. The approach does not by itself offer a mechanism for specifying the order between aspects, but it inherits the mechanisms of the underlying AOP languages.

Sharing: Besides adopting the features for re-using aspect offered by the underlying language, the Framed Aspects approach itself provides re-usable aspect templates. The templates can be concretized through the configuration file.

Commonly, the presented approaches allow explicit declaration of precedence among aspects which can address ordering constraints between instrumentations. AspectJ and CaesarJ support re-use (i.e., sharing) and configuration (i.e., scoping) through inheritance and overriding; in HyperJ code can be re-used by composition while the composition specification (the control file) cannot be re-used at all. JAsCo and Composition Filters support black-box re-use and configuration through parameterization, however, the languages only support parameters for a limited set of constructs. These two approaches and CaesarJ additionally provide control over scoping by means of their programmatic deployment or superimposition features. Framed Aspects supports parameters in a more flexible way, but parameterization requires a fair amount of variability analysis to determine extension points of a planned feature. Since it requires predetermination of extension points, it limits the use of unforeseen instrumentations.

4 Conclusion

Although tools based on instrumentation techniques are in wide-spread use, the engineering of such tools often violates basic reuse principles. As efficiency of the tools is of paramount importance, low-level instrumentation frameworks, which suffer from a lack of mechanisms for refining and composing instrumentations, are commonly used. As a consequence, instrumentations are often implemented by resorting to the tedious and error-prone copy/paste anti-pattern.

In this position paper we identified two challenges that need to be addressed by future mechanisms in support of refinement and composition of instrumentations: specification and enforcement of constraints, and avoidance of hard-coded dependencies. We illustrated these challenges with a running example.

In our ongoing research, we are exploring novel refinement and composition mechanisms in the context of the domain-specific instrumentation language DiSL [9,18]. We are working on instrumentation contracts that make constraints explicit and allow for automated checks that enforce these constraints. Furthermore, we are integrating a mechanism for dependency injection to deal with the problem of hard-coded dependencies.

Acknowledgments. The research presented here was conducted while L. Marek was with the University of Lugano. It was supported by the Scientific Exchange Programme NMS–CH (project code 10.165), by a Sino-Swiss Science and Technology Cooperation (SSSTC) Institutional Partnership (project no. IP04–092010), by the Swiss National Science Foundation (project CRSII2_136225), by the Science and Technology Commission of Shanghai Municipality (project no. 11530700500), by the Czech Science Foundation (project GACR P202/10/J042), as well as by the German Federal Ministry of Education and Research (BMBF) in EC SPRIDE and by the Hessian LOEWE excellence initiative in CASED. The authors thank Éric Tanter for comments on a draft of this paper.

References

1. Ammons, G., Ball, T., Larus, J.R.: Exploiting hardware performance counters with flow and context sensitive profiling. In: Proceedings of the Conference on Programming Language Design and Implementation, pp. 85–96 (1997)
2. Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K.: An Overview of CaesarJ. In: Rashid, A., Aksit, M. (eds.) Transactions on AOSD I. LNCS, vol. 3880, pp. 135–173. Springer, Heidelberg (2006)
3. Bergmans, L.M.J.: Akşit, M.: Principles and design rationale of composition filters. In: Aspect-Oriented Software Development, pp. 63–96. Addison-Wesley (2004)
4. Binder, W., Ansaloni, D., Villazón, A., Moret, P.: Flexible and efficient profiling with aspect-oriented programming. Concurrency and Computation: Practice and Experience 23(15), 1749–1773 (2011)
5. Chiba, S.: Load-Time Structural Reflection in Java. In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, pp. 313–336. Springer, Heidelberg (2000)
6. Flanagan, C., Freund, S.N.: The RoadRunner dynamic analysis framework for concurrent programs. In: Proceedings of the 9th Workshop on Program Analysis for Software Tools and Engineering, pp. 1–8 (2010)
7. Greenwood, P., Blair, L.: A Framework for Policy Driven Auto-Adaptive Systems Using Dynamic Framed Aspects. In: Rashid, A., Aksit, M. (eds.) Transactions on AOSD II. LNCS, vol. 4242, pp. 30–65. Springer, Heidelberg (2006)
8. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Lee, S.H. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–354. Springer, Heidelberg (2001)
9. Marek, L., Villazón, A., Zheng, Y., Ansaloni, D., Binder, W., Qi, Z.: DiSL: a domain-specific language for bytecode instrumentation. In: Proceedings of the 11th International Conference on Aspect-Oriented Software Development (2012)
10. Moret, P., Binder, W., Tanter, É.: Polymorphic bytecode instrumentation. In: Proceedings of the 10th International Conference on Aspect-Oriented Software Development, pp. 129–140 (2011)
11. Rudolph, J., Thiemann, P.: Mnemonics: type-safe bytecode generation at run time. In: Proceedings of the Workshop on Partial Evaluation and Program Manipulation, pp. 15–24 (2010)
12. Suvée, D., Vanderperren, W., Jonckers, V.: JAsCo: an aspect-oriented approach tailored for component based software development. In: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, pp. 21–29 (2003)

13. Tanter, E., Moret, P., Binder, W., Ansaloni, D.: Composition of dynamic analysis aspects. In: Proceedings of the 9th International Conference on Generative Programming and Component Engineering, pp. 113–122 (2010)
14. Tarr, P., Osher, H., Stanley, M., Sutton, J., William Harrison, W.: N degrees of separation: multi-dimensional separation of concerns. In: Aspect-Oriented Software Development, pp. 37–61. Addison-Wesley (2004)
15. Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java optimization framework. In: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, pp. 214–224 (1999)
16. Villazón, A., Binder, W., Moret, P.: Flexible calling context reification for aspect-oriented programming. In: Proceedings of the 8th International Conference on Aspect-Oriented Software Development, pp. 63–74 (2009)
17. Villazón, A., Binder, W., Moret, P., Ansaloni, D.: Comprehensive aspect weaving for Java. Science of Computer Programming 76(11), 1015–1036 (2011)
18. Zheng, Y., Ansaloni, D., Marek, L., Sewe, A., Binder, W., Villazón, A., Tuma, P., Qi, Z., Mezini, M.: Turbo DiSL: partial evaluation for high-level bytecode instrumentation. In: TOOLS 2012 – Objects, Models, Components, Patterns (2012)

Constructing Customized Interpreters from Reusable Evaluators Using Game

Stijn Timbermont, Coen De Roover, and Theo D'Hondt

Vrije Universiteit Brussel
{stimberm,cderoove,tjdhondt}@vub.ac.be

Abstract. Separation of concerns is difficult to achieve in the implementation of a programming language interpreter. We argue that evaluator concerns (i.e., those implementing the operational semantics of the language) are, in particular, difficult to separate from the runtime concerns (e.g., memory and stack management) that support them. This precludes the former from being reused and limits variability in the latter.

In this paper, we present the Game environment for composing customized interpreters from a reusable evaluator and different variants of its supporting runtime. To this end, Game offers a language for specifying the evaluator according to the generic programming methodology. Through a transformation into defunctionalized monadic style, the Game toolchain generates a *generic abstract machine* in which the sequencing of low-level interpretational steps is parameterized. Given a suitable instantiation of these parameters for a particular runtime, the toolchain is able to inject the runtime into the generic abstract machine such that a complete interpreter is generated.

To validate our approach, we port the prototypical Scheme evaluator to Game and compose the resulting generic abstract machine with several runtimes that vary in their automatic memory management as well as their stack discipline.

1 Introduction

In the implementation of an *interpreter* for a programming language, one can distinguish evaluator concerns from the runtime concerns that support them. The *evaluator concerns* implement the operational semantics of the language, often assuming computational resources are unbounded. The *supporting runtime concerns* maintain this illusion on a physical machine through data structures and algorithms for memory and stack management.

There is great variation among an interpreter's runtime support, of which we identify the following sources:

- The first source of variation is the evaluator itself. Different language features require different kinds of runtime support. For instance, closures require the ability to capture an environment and keep it alive for an indeterminate period of time. Likewise, exception handling requires the ability to skip computations after an error occurred and to proceed with the exception handler.

T. Gschwind et al. (Eds.): SC 2012, LNCS 7306, pp. 97–113, 2012.

These features affect the implementation of environments and of the execution stack respectively.
- The second source of variation is the host platform. If the host platform is a high-level language, it will also have a supporting runtime that can be reused. For instance, when implementing a garbage collected language in another garbage collected language. The bigger the mismatch between the host platform and the evaluator, however, the more effort is required from the supporting runtime. This is, for instance, the case when implementing a language with first-class continuations in C.
- The non-functional requirements of the interpreter represent the third source of variation. The supporting runtime must not only satisfy the needs of the evaluator, it must also do so in a manner that meets efficiency requirements (e.g., memory usage and power consumption). This leads to many trade-offs, which are influenced by the idiosyncrasies of the host platform as well as the expected usage of the interpreter (i.e., the kinds of programs it will run).

On par with the operational semantics of the language, the evaluator should not be affected by variation in runtime support. However, the practice of interpreter development does not reflect this. In the implementation of an interpreter, evaluator concerns are difficult to separate from the runtime concerns that support them. This precludes the former from being reused and limits variability in the latter. The contributions of this paper are as follows:

- Using the case of automatic memory management, we illustrate that the choice for a particular runtime has a severe impact on the structure of the evaluator (Section 2).
- We introduce the notion of a *generic abstract machine* (Section 3.2), an abstract machine [5] that anticipates the supporting runtime without committing to any details using generic programming techniques. A generic abstract machine corresponds to the evaluator in an intermediate form called *defunctionalized monadic style*.
- We present the Game environment (Section 3), consisting of a language and a toolchain centered around the notion of a *generic abstract machine*. Using the Game programming language, the developer of the evaluator decides on the interface between the evaluator and the runtime —thus enabling reuse of the evaluator. Using the Game toolchain, the developer of the supporting runtime can inject a concrete runtime variant in the evaluator —giving rise to a customized interpreter. Our proof-of-concept implementation of the Game toolchain generates this interpreter in a subset of R6RS Scheme. This subset is sufficiently low-level for targeting C to be realistic.
- We validate our approach by implementing the prototypical Scheme evaluator from SICP [1] in Game (Section 3.1) and instantiating the resulting *generic abstract machine* with several runtimes (Section 4). These runtimes vary in their memory management (e.g., a non-moving mark-and-sweep versus a moving stop-and-copy GC) and in their stack discipline (e.g., reusing the host stack versus managing an explicit stack). The generated

interpreters are, together with the GAME prototype, publicly available at http://soft.vub.ac.be/~stimberm/game/sc12/.

2 Motivating Example: The Impact of Automatic Memory Management on the Structure of an Evaluator

This section illustrates the impact of runtime support on the structure of the evaluator. Their interaction inhibits reusing the evaluator in other interpreters.

Consider adding automatic memory management to the prototypical evaluator for Scheme depicted in Figure 1. The evaluator corresponds to the one from Section 4.1 of SICP [1], implemented in GAME (cf. Section 3.1). For the purpose of this section, GAME has the same syntax and semantics as regular Scheme.

Garbage collectors (GCs) use reachability as a heuristic to determine whether an object on the heap can be reclaimed. It is the evaluator's responsability to hand the GC all heap objects that are a priori reachable. Objects that cannot be reached from these *root pointers* can no longer be accessed from the program and are therefore safe to reclaim. Root pointer treatment comprises the main source of interaction between an evaluator and a GC.

Varying an interpreter's GC strategy may require varying the treatment of root pointers in the interpreter's evaluator. Some GC algorithms move objects around to avoid fragmentation. All pointers to a particular object must be updated to reflect the new location of the object. Concretely, the evaluator should not use the pointers stored in local variables after every point in its execution where GC may have occurred.

The consequences of errors in the treatment of root pointers are severe. If the evaluator neglects to communicate a root pointer to the GC, the object referred to by the root pointer may be reclaimed. If the evaluator subsequently dereferences this pointer again, the resulting behavior is unpredictable. The memory chunk may have been cleared, or it may have been reused to store another object. After a *moving* GC, a neglected root pointer may even point *in the middle* of some other object. Such bugs are difficult to diagnose. Their occurrence depends on the state of the entire heap and on the arbitrary moment GC occurs.

Restructuring the SICP Evaluator for Garbage Collection. As the GC expects to be handed a set of root pointers, we have to restructure the entire SICP evaluator such that it can construct this set at all places where a GC might occur. Consider the `list-of-values` function depicted on line 35 of Figure 1. It returns a list that contains the values to which the expressions `exps` evaluate one by one in the environment `env`. As evaluating an individual expression may trigger a GC, there is a risk of dangling pointers within this function. We will adapt its code such that root pointers are preserved for a non-moving and a moving GC respectively.

Adaptation 1: Non-Moving GC. In Figure 2a, recursive evaluations within the function have been made explicit. In this version, it is clear that variable `first` may contain a root pointer to an object on the heap, which must be preserved

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))

(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))

(define (list-of-values exps env)
  (if (no-operands? exps)
      null
      (cons (eval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (eval (assignment-value exp) env)
                       env)
  ok-symbol)

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (eval (definition-value exp) env)
                    env)
  ok-symbol)

(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))

(define (eval-sequence exps env)
  (if (last-exp? exps)
      (eval (first-exp exps) env)
      (begin (eval (first-exp exps) env)
             (eval-sequence (rest-exps exps) env))))
```

Fig. 1. The prototypical Scheme evaluator from Section 4.1 of SICP [1]

```
(define (list-of-values exps env)
  (if (no-operands? exps)
      null
      (let ((first (eval (first-operand exps) env))
            (rest  (list-of-values (rest-operands exps) env)))
        (cons first rest))))
```

(a) with explicit names for subcomputations

```
(define (list-of-values exps env)
  (if (no-operands? exps)
      null
      (begin (register-nm exps)
             (register-nm env)
             (define first (eval (first-operand exps) env))
             (register-nm first)
             (define rest (list-of-values (rest-operands exps) env))
             (unregister-nm 3)
             (cons first rest))))
```

(b) with root registration for a non-moving GC

```
(define (list-of-values exps env)
  (if (no-operands? exps)
      null
      (begin (register-m exps)
             (register-m env)
             (define first (eval (first-operand exps) env))
             (define exps2 (unregister-m))
             (define env2  (unregister-m))
             (register-m first)
             (define rest (list-of-values (rest-operands exps2) env2))
             (define first2 (unregister-m))
             (cons first2 rest))))
```

(c) with root registration for moving GC

Fig. 2. Adaptations of `list-of-values`

across subsequent evaluations triggered by the recursive call to `list-of-values`. Variables `exps` and `env` may also contain root pointers as expressions and environments can be stored on the heap. Figure 2b depicts function `list-of-values` as adapted to a particular non-moving GC. Root pointers are registered with the GC through `register-nm` and are subsequently unregistered through `unregister-nm`. These functions behave in a LIFO manner. The parameter to `unregister-nm` indicates that 3 pointers must be discarded from the root set. Note that `rest` does not have to be registered as it is passed to `cons` without any interleaved evaluation.

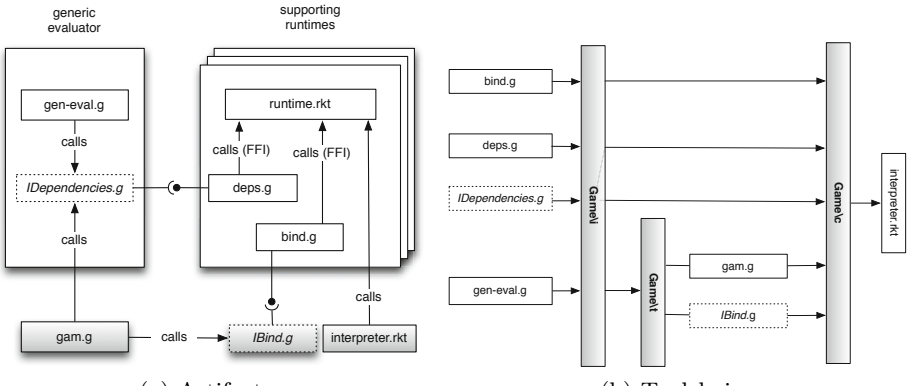
Adaptation 2: Moving GC. A moving GC further complicates the treatment of root pointers. If a local variable contains a root pointer prior to a potential GC, merely registering that root pointer no longer suffices. The GC may move the object around on the heap and render the pointer stored in the local variable invalid. This problem can be solved by having the GC provide the new root locations to the evaluator, which must then refrain from using the old pointers. Figure 2c depicts function `list-of-values` as adapted to such a moving GC. Root pointers are registered with the GC through `register-m`. In turn, the GC provides

the updated root pointer through a corresponding call to `unregister-m`. Again, these functions behave in a LIFO manner. Note that a root pointer stored in a local variable is never used again after a recursive evaluation.

The two adaptations show that composing the SICP evaluator with a custom GC has a structural impact on the evaluator. Furthermore, the details of the restructuring depends on the chosen GC strategy, which means that it is also not possible to prepare the evaluator for GC once and for all. Instead, the evaluator must be adapted for each variation of memory management. This lack of separation of concerns hinders reuse of the evaluator and evolution of the interpreter as a whole.

3 Overview of Game

The novel notion of a *generic abstract machine* is key to our approach to constructing customized interpreters from a reusable evaluator and different variants of a supporting runtime. A generic abstract machine corresponds to a recursive evaluator implemented using generic programming techniques such that a runtime is anticipated but not yet committed to, transformed into a low-level form that lends itself better to injecting a concrete runtime. To support this approach, we developed Game; the Generic Abstract Machine Environment. Using Game, constructing an interpreter entails four different activities: one for the evaluator developer, another for the runtime developer and two that are left to the toolchain. Before discussing these activities in detail, we briefly outline their key aspects using Figure 3. Figure 3a clarifies the interdependencies of the different kinds of artifacts that are involved in these activities and indicates whether they are generated or have to be provided by one of the developers. Figure 3b depicts how these artifacts flow throughout the environment.



(a) Artifacts (b) Toolchain

Fig. 3. Overview of Game

Developing a generic evaluator (evaluator developer). GAME offers the generic programming language GAME\l for implementing an evaluator in a high-level, recursive functional style. Most importantly, this evaluator does not have to adhere to the specifics that would be dictated by the choice for a particular supporting runtime (e.g., the root pointer treatment protocols discussed in Section 2). The evaluator's own dependencies on the runtime (i.e., operations that must be provided by every runtime it will be composed with) are specified through interface declarations. In Figure 3a, `gen-eval.g` and `IDependencies.g` correspond to the generic evaluator and the interface declaration of its dependencies respectively. Note that both artifacts are the sole responsibility of the evaluator developer.

Deriving a generic abstract machine (GAME). Given such a generic evaluator, GAME's transformation engine GAME\t transforms it into an abstract machine —inspired by the work on defunctionalized interpreters [4,2]. The actual transformation inserts hooks that make every computational step in the generic evaluator explicit and programmable. As these hooks are declared through an interface, we call the result a *generic abstract machine*. The transformation requires type information computed for GAME\l input by a type inferencer called GAME\i. In Figure 3a, `gam.g` and `IBind.g` correspond to the generic abstract machine and the interface for the hooks respectively. Note that the generic abstract machine inherits the dependencies of the original evaluator, hence the arrow from `gam.g` to `IDependencies.g`. Figure 3b indicates that `gam.g` is produced from `gen-eval.g` by GAME\t.

Instantiating the generic abstract machine (runtime developer). It is now up to the runtime developer to inject a concrete runtime into the generic abstract machine that was generated above. This will yield a customized interpreter expressed in GAME\l. Note that the runtime developer does not have to analyze the generic abstract machine itself, but only has to implement the interface dependencies of the machine. In other words, `gam.g` can be considered a black box. In Figure 3a, `deps.g` implements the interface of the evaluator dependencies declared in `IDependencies.g`; `bind.g` implements the interfaces for the additional dependencies of the generic abstract machine declared in `IBind.g`. Both implementations import runtime functionality from `runtime.rkt` using a simple Foreign-Function Interface. Developing these three files is the sole responsibility of the runtime developer.

Generating an executable interpreter (GAME). The final activity entails compiling the GAME\l interpreter constructed above to the host platform. In this case, GAME's compiler GAME\c targets R6RS Scheme, in particular Racket[1]. GAME\c performs simple optimizations, converts polymorphic into monomorphic code and removes the overhead introduced by the generic programming. Furthermore, the generated code does not use Scheme's more advanced features such as dynamic typing, long-lived closures, first-class continuations or tail-call elimination in such a way that C code generation would

[1] `http://racket-lang.org/`

```
(define (true? x)
  (not (eq? x false_sv)))
(define (self-evaluating? exp)
  (cond ((number? exp) true)
        ((string? exp) true)
        (else false)))
(define (tagged-list? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
      false))
(define (lambda? exp)
  (tagged-list? exp lambda-symbol))
(define (lambda-parameters exp)
  (cadr exp))
(define (lambda-body exp)
  (cddr exp))
```

(a) Fragment of the helper functions implemented in Game\l

```
(type-function (SV) *)

(interface (Scheme)
 (ok-symbol (tf SV))
 ...
 (true_sv  (tf SV))
 (false_sv (tf SV))
 ...
 (pair?   (-> ((tf SV)) (effect) Bool))
 (number? (-> ((tf SV)) (effect) Bool))
  ...
 (car (-> ((tf SV)) (effect IO) (tf SV)))
 ...
 (cons (-> ((tf SV) (tf SV))
           (effect IO GC)
           (tf SV)))
 ...)
```

(b) Interface for the required dependencies of the SICP evaluator

Fig. 4. Supporting functions for the SICP evaluator

be unfeasible. Figure 3b illustrates how Game\c compiles the generated `interpreter.rkt` together with `runtime.rkt` into a final executable interpreter.

3.1 Developing a Generic Evaluator

Figure 1 depicts the prototypical Scheme evaluator from SICP [1] implemented in Game\l. Other than substituting `ok-symbol` for the Scheme symbol `'ok`, the code is identical to the original. The evaluator relies on various helper functions, such as `variable?` and `lambda?` for testing whether an expression is of a certain type, and `lookup-variable-value` and `extend-environment` for manipulating environments. Some of these helper functions can also be implemented in Game\l itself (see Figure 4a). However, at some point the evaluator requires functions such as `number?` and `car` which are not available in Game\l. Enumerating these functions as required dependencies renders the evaluator *generic*.

Figure 4b depicts an extract from the Game\l declaration of the required dependencies for the generic SICP evaluator, using an `interface` (akin to a Haskell type class). Note that the required dependencies are explicitly typed. We introduce a single type `sv` for typing Scheme values as Scheme is dynamically typed. Because the concrete type depends on the runtime the evaluator is instantiated with, we define `sv` as a type function [16], without any arguments. The `*` indicates the kind of `sv`. Types can use this type function using `(tf sv)`. For example, the type for `ok-symbol` is simply `(tf sv)`, because symbols are regular Scheme values. The type for the function `pair?` is `(-> ((tf sv)) (effect) Bool)`, where `->` is a type constructor with three arguments; first a list of types for the parameters of the function; third the result type of the function; second the side-effects of the function. Section 3.2 explains the crucial role these effect annotations play.

3.2 Deriving a Generic Abstract Machine

As illustrated in Section 2, the original SICP evaluator has to be restructured before it can be composed with a custom GC. Furthermore, there are variations in the details of the restructuring for a moving and a non-moving GC. Game introduces defunctionalized monadic style to capture the essence of this restructuring and to abstract over the details. As its name suggests, defunctionalized monadic style derives from monadic style, which generalizes continuation-passing style. Computations are explicitly sequenced, using higher-order functions to represent continuations. Defunctionalized monadic style turns these continuations into explicit data structures, such that the transfer of data between sequenced computations also becomes explicit. Therefore, an evaluator written in defunctionalized monadic style has a more low-level structure than a recursive evaluator.

Figure 5 depicts function `list-of-values` (cf. Figure 1) in defunctionalized monadic style. The essential construct in defunctionalized monadic style is special operator `>>>` (pronounced *bind*). It explicitly sequences two computations, of which the first is passed as the first argument. The second and third parameters represent the second computation. The second parameter is a reference to a continuation function, a top-level function that receives the result of the evaluation. The third argument lists all additional data that should be preserved because it is still needed in the continuation function. The continuation function receives two arguments: first the preserved data, then the result of the evaluation. In the example of `list-of-values`, the first expression in `exps` is evaluated, but both `exps` and `env` are preserved for later, when they are needed in the continuation function `cnt-list-of-values-1`. This is achieved by passing the tuple `(* exps env)` as third argument to `>>>`. That continuation function `cnt-list-of-values-1` uses pattern matching, a Game\l feature, on its first argument to retrieve `exps` and `env` again. The second argument `first` is the result of evaluating the first operand. The body of `cnt-list-of-values-1` uses `>>>` again to sequence the evaluation of the rest of the arguments with `cnt-list-of-values-2`. This time, only `first` must be preserved. Finally, `cnt-list-of-values-2` simply combines the results using `cons`.

An evaluator expressed in defunctionalized monadic style has the structure of an abstract machine [5], and is better suited for composition with a supporting runtime. Using Game, abstract machines do not have to be developed by hand.

```
(define (list-of-values exps env)
  (if (no-operands? exps)
      null
      (>>> (eval (first-operand exps) env)
           cnt-list-of-values-1
           (* exps env))))
(define (cnt-list-of-values-1 (* exps env) first)
  (>>> (list-of-values (rest-operands exps) env)
       cnt-list-of-values-2
       (* first)))
(define (cnt-list-of-values-2 (* first) rest)
  (cons first rest))
```

Fig. 5. The function `list-of-values` in defunctionalized monadic style

```
(defmacro (>>> e k frame)
  (bind (lambda () e) k frame))

(interface (Frame (frm *) (t *) (res *))
  (bind (forall ((e !*) (ke !*))
          (-> (  (-> () e t)
                 (-> (frm t) ke res)
                 frm)
              (effect IO e ke)
              res))))
```

Fig. 6. Declaration of >>> and bind

Instead, Game derives them from evaluators by transforming the latter to defunctionalized monadic style. This transformation is referred to as Game\t and follows the work of defunctionalized interpreters [4,2], which in turn goes back to the seminal work on definitional interpreters [13]. The essence of their derivation is CPS transformation followed by defunctionalization, yielding first-order, tail-recursive evaluators, which are equivalent to well-known abstract machines. Game\t works similarly, but adapts the original in two ways: first, Game\t generates a *generic* abstract machine, and second, Game\t works *automatically*, whereas in [13,2], the derivation is performed by hand. These two adaptations are further discussed in the following two paragraphs, respectively.

Generic Abstract Machines. A generic abstract machine is expressed in defunctionalized monadic style, where the semantics of >>> is customizable, that is, >>> is a hook in the abstract machine, used to inject the supporting runtime (see Section 3.3). This is achieved by using generic programming, in the same way that the evaluator dependencies are treated. However, >>> is not a regular function because it should not execute its first argument immediately; instead, we define >>> as a macro in terms of a function bind, by wrapping the first argument to >>> in a thunk, as shown in Figure 6. The function bind is overloaded on the types of the frame, the value produced by the thunk and the final result of the continuation. The type of bind is polymorphic in the effects of the higher-order arguments, hence the effect variables e and ke with kind !*.

Automatic Derivation. It is not necessary to transform every function call in the evaluator to defunctionalized monadic style. Helper functions such as number? and car do not trigger a GC, so introducing a >>> construct would be overkill. Therefore, Game\t applies the transformation selectively: only computations that potentially trigger a GC are transformed into defunctionalized monadic style. To distinguish between computations that may trigger a GC and those who do not, Game\t uses information derived from a type and effect system in the style of [18]. Effect annotations are useful to track properties such as "may trigger a GC" because they propagate through the evaluator: a function that calls another function which potentially triggers a GC inherits this property. Using Game, there is no need to manually annotate an evaluator with effects. Instead, Game\i performs type and effect inference which derives the required information from unannotated code. However, the declaration of the

dependencies must explicitly declared their type as they cannot be "guessed" by Game\i. Figure 4b shows some of the type annotations. Function types not only specify the types of the arguments and result, but also the effect of the function. An effect `(effect ...)` denotes the union of elementary effects. For example, `(effect)` denotes the empty effect and `(effect IO)` the singleton effect, where `IO` indicates general side effects such as accessing and modifying heap-allocated memory. The type of `cons` is special: the effect of the function is `(effect IO GC)`, which means that `cons` may produce both the general effect `IO` and the effect `GC`. This annotation is propagated through the evaluator and is used by the transformation to distinguish between functions that may trigger a GC and those who do not.

These two adaptations, with the combination of Game\i with Game\t, forms the basis for the automatic derivation of generic abstract machines from evaluators. The type annotation for `cons` in Figure 4b indicates that invocations of `cons` are a source of GC. This information is propagated through the unannotated evaluator, such that Game\i assigns the following type to `eval`.

```
(forall () (Scheme)
        (-> ((tf SV) (tf SV)) (effect IO GC) (tf SV)))
```

The effect annotation `(effect IO GC)` indicates that `eval` may trigger a GC. The type constraint `Scheme` accounts for the required dependencies used in the evaluator. Subsequently, Game\l introduces `>>>` only for those computations that include `GC` in their effect. This is reflected in the following new type of eval.

```
(forall () (Scheme
            (Frame (*                  ) (tf SV) (tf SV))
            (Frame (* (tf SV)          ) (tf SV) (tf SV))
            (Frame (* (tf SV) (tf SV)) (tf SV) (tf SV)))
        (-> ((tf SV) (tf SV)) (effect IO GC) (tf SV)))
```

The three `Frame` constraints arise because `>>>` is overloaded on the type of the frame (via `bind`). For example, the constraint `(Frame (* (tf SV) (tf SV)) (tf SV) (tf SV))` indicates that at some point a `>>>` construct is used to preserve two Scheme values.

3.3 Instantiation of Generic Abstract Machines

In this section we illustrate how we can instantiate a generic abstract machine by providing implementations for both the required dependencies of the original evaluator and the additional dependencies introduced by the derivation of the generic abstract machine. In this section we only give a "default" instantiation to explain the mechanism in Game. The more interesting instantiations for GC are given in Section 4.

Figure 7 gives the Game\l code that instantiates the generic machine produced by Game\t. As the type of `eval` indicates, this instantiation must satisfy four constraints, one `Scheme` constraint and three `Frame` constraints. In Figure 7a, we use a Foreign-Function Interface (FFI) to import the appropriate functions from the underlying platform, in this case the R6RS Scheme implementation of PLT Racket. We also introduce a single type `RV` denoting a Racket value. In Figure 7b, we use `axiom` constructs (akin to Haskell's type class instances) to actually

```
(type-decl RV () *)
(foreign-import "pair?" r_pair?
  (-> (RV) (effect) RV))
(foreign-import "number?" r_number?
  (-> (RV) (effect) RV))
...
(foreign-import "car" r_car
  (-> (SV) (effect IO) SV))
(foreign-import "cdr" r_cdr
  (-> (SV) (effect IO) SV))
...
(define (bindR thunk k frame)
  (k frame (thunk)))
```

(a) FFI imports and definitions

```
(axiom () () (~ (tf SV) RV))
(axiom () ()
       (? Scheme)
  (mkScheme ...
            r_pair?
            r_number?
            ...
            r_car
            r_cdr
            ...))
(axiom ((frm *)) ()
       (? (Frame frm RV RV))
  (mkFrame bindR))
```

(b) Instantiation with `axiom`

Fig. 7. Default instantiation of the SICP generic abstract machine

instantiate the generic abstract machine by satisfying its dependencies. The type function `SV` is instantiated with `RV` and the `Scheme` constraint with the imported functions. The three `Frame` constraints are all instantiated with `bindR`, which simply executes `thunk` and passes the resulting value along with the frame to the continuation. Note that this implementation introduces no supporting runtime, and simply reintroduces the recursion that was made explicit by GAME\t. The `axiom` declaration for `Frame` is polymorphic in the type of the frame, which is why there is only one.

4 Evaluation

To evaluate GAME we instantiate the generic abstract machine derived from the SICP evaluator with several runtimes, which vary in their memory management and stack discipline. Concretely, we give three instantiations: one for a non-moving mark-and-sweep and another for a moving stop-and-copy GC which both rely on the recursion stack of the underlying platform (Racket); and finally an instantiation for a custom stack, using trampolining. We stress again that all three instantiations reuse the original evaluator from Figure 1.

Instantiation for a Non-Moving Mark-and-Sweep GC. For this instantiation, the implementation for `cons`, imported via the FFI in Figure 7a, is not the standard Racket `cons`, but a custom implementation which uses a mark-and-sweep GC. The code for the instantiation itself is shown in Figure 8b. Figure 8a defines helper functions for the registration of root pointers in frames, which ultimately rely on `register-nm` and `unregister-nm` which were also used in Section 2. These functions are part of the mark-and-sweep memory manager and are also imported using the FFI. In Figure 8b, `bindNonMovGC` satisfies the `Frame` constraints, by registering the roots in the frame before executing `thunk`, then unregistering the corresponding number of roots and finally proceeding with the continuation function. The `axiom` for `Frame` is again polymorphic, by referring to the `NMRoots` interface, because `bindNonMovGC` is defined for every frame that supports `register-roots-nm`.

```
(interface (NMRoots (t *))
  (register-roots-nm
      (-> (t) (effect IO) Int)))

(foreign-import "register-nm" register-nm
  (-> (RV) (effect IO) Unit))
(foreign-import "unregister-nm" unregister-nm
  (-> (Int) (effect IO) Unit))

(define (register0 (*)) 0)
(define (register1 (* val))
  (register-nm val)
  1)
(define (register2 (* val1 val2))
  (register-nm val1)
  (register-nm val2)
  2)
```

(a) Non-moving root registration

```
(define (bindNonMovGC thunk frame k)
  (let* ((cnt (register-roots-nm frame))
         (val (thunk)))
    (unregister-nm cnt))
    (k frame val))

(axiom ((frm *)) ((? (NMRoots frm)))
       (? (Frame frm RV RV))
  (mkNMRoots bindNonMovGC))
(axiom () () (? (NMRoots (*)))
  (mkNMRoots register0))
(axiom () () (? (NMRoots (* RV)))
  (mkNMRoots register1))
(axiom () () (? (NMRoots (* RV RV)))
   (mkNMRoots register2))
```

(b) Instantiation

Fig. 8. Instantiation for a non-moving mark-and-sweep GC

Instantiation for a Moving Stop-and-Copy GC. For a moving GC, the instantiation with `bindNonMovGC` in Figure 8b is not suitable, because the continuation function receives the old `frame`, whose root pointers may be invalidated if a moving GC occurs in the course of executing `thunk`. Figure 9 shows an adapted instantiation. The roots are treated differently, using the function `register-m` and `unregister-m`, which work in a LIFO manner (see Section 2). The function `bindMovGC` in Figure 9b registers to roots before executing the `thunk` and then unregistering them again, which gives the update frame `frame2`, which is then passed to `k`. Note that the original `frame` is not used after registering it.

Instantiation for an Explicit Stack. Figure 10 shows an instantiation which maintains an explicit recursion stack instead of reusing the host stack. The definition of `bindStack` in Figure 10b pushes both the continuation and the frame on the stack and simply returns the value of executing the thunk. The function `engine` defines a loop which pops the top continuation of the stack and executes it until the stack is empty. The continuation pushed on the stack in `bindStack` is not the original `k`. Instead, it is wrapped in a `lambda` such that it pops its own frame from the stack, as the layout of the frame may vary across continuation functions. Note however that `k` is a reference to a top-level function and that the `lambda` expression does not capture other local variables. This means that if GAME\c inlines `bindStack`, the `lambda` can be lifted, such that again only top-level function pointers are required from the host platform. The functions `push0`, `pop0`, ..., are similar to the `register` functions of the previous instantiation, but instead they ultimately rely on `push-rv` and `pop-rv`. This instantiation effectively converts the evaluator into trampolined style [7]. Therefore, this instantiation would be a good starting point for extending the interpreter with first-class continuations.

Discussion. All three instantiations[2] successfully compile to Racket and correctly execute a small Scheme program. In particular, the GC instantiations do

[2] The full code of the generated interpreters can also be found on `http://soft.vub.ac.be/~stimberm/game/sc12/`

```
(interface (MRoots (t *))
  (register-roots-m
      (-> (t) (effect IO) Unit))
  (unregister-roots-m
      (-> () (effect IO) t)))

(foreign-import "register-m" register-m
  (-> (RV) (effect IO) Unit))
(foreign-import "unregister-m" unregister-m
  (-> () (effect IO) RV))

(define (register0 (*))  unit)
(define (unregister0)    (*))
(define (register1 (* val))
  (register-m val))
(define (unregister1)
  (* (unregister-m)))
(define (register2 (* val1 val2))
  (register-m val1)
  (register-m val2))
(define (unregister2)
  (let* ((val2 (unregister-m))
         (val1 (unregister-m)))
    (* val1 val2)))
```

(a) Moving root registration

```
(define (bindMovGC thunk frame k)
  (register-roots-m frame)
  (let* ((val    (thunk))
         (frame2 (unregister-roots-m)))
    (k frame2 val)))

(axiom ((frm *)) ((? (MRoots frm)))
       (? (Frame frm RV RV))
  (mkFrame bindMovGC))
(axiom () () (? (MRoots (*)))
  (mkMRoots register0 unregister0))
(axiom () () (? (MRoots (* RV)))
  (mkMRoots register1 unregister1))
(axiom () () (? (MRoots (* RV RV)))
   (mkMRoots register2 unregister2))
```

(b) Instantiation

Fig. 9. Instantiation for a moving stop-and-copy GC

not suffer from dangling or corrupt pointers after a GC. Manual instantiations would require three rewrites of the SICP evaluator, whereas by using Game, the evaluator can be reused as is. However, Game also has a number of limitations. First of all, an instantiation of a generic abstract machine consists of a single monolithic entity, which must be composed manually by the runtime developer. For example, code that does not go through the Game toolchain still has to track roots itself. Second, the evaluator is treated as a single entity, with little support for modularization, especially compared to modular interpreters using monad transformers [11,17]. However, we believe both of these concerns to be orthogonal to the original goal of Game, which is to separate concerns *between* and not *whitin* the evaluator and the runtime.

5 Related Work

In the context of modularity in interpreters, monads and monad transformers proved to be a very useful abstraction technique [11,17]. The notion of defunctionalized monadic style incorporates the abstraction power of monadic style. Furthermore, both approaches share the use of type classes to express the monadic operators and vary their behavior. However, the work on monadic interpreters focussed on modularization of individual language constructs, without considering the supporting runtime. Game on the other hand, specifically considers the supporting runtime, and its relation with the evaluator as a whole.

PyPy [14] is both a toolchain for implementing virtual machines in RPython (a restricted subset of Python) and a meta-circular implementation of Python (in RPython). PyPy translates virtual machines written in RPython to C, and in the process also injects required runtime functionality. PyPy supports variations of the memory management strategy: the default translation uses a conservative

```
(foreign-import "push-rv" push-rv
  (-> (RV) (effect IO) Unit))
(foreign-import "pop-rv" pop-rv
  (-> () (effect IO) RV))

(foreign-import "pushCnt" pushCnt
  (forall ((ke !*))
    (-> ((-> (RV) ke RV))
        (effect IO)
        Unit)))
(foreign-import "popCnt" popCnt
  (-> () (effect IO) (-> (RV)
                         (effect IO)
                         RV)))
(foreign-import "stack-empty?"
                stack-empty?
  (-> () (effect IO) Bool))
```

(a) Pushing and popping values and continuation functions

```
(define (bindStack thunk frame k)
  (push frame)
  (pushCnt (lambda (val) (k (pop) val)))
  (thunk))

(define (engine val)
  (if (stack-empty?)
      val
      (engine ((popCnt) val))))

(axiom ((frm *)) ((? (Stack frm)))
       (? (Frame frm RV RV))
  (mkFrame bindStack))
(axiom () () (? (Stack (*)))
  (mkStack push0 pop0))
(axiom () () (? (Stack (* RV)))
  (mkStack push1 pop1))
(axiom () () (? (Stack (* RV RV)))
  (mkStack push2 pop2))
```

(b) Instantiation

Fig. 10. Instantiation for an explicit stack

GC but it is also possible to choose a mark-and-sweep GC. However, in PyPy, the GC must be specified in relation to RPython, whereas Game allows the GC to be tailored towards the implemented evaluator.

The combination of transformation into continuation-passing style and defunctionalization (transforming the continuations into data structures) was introduced in [13] and extensively used to relate evaluators and abstract machines [2,4], but also to restructure programs for the Web [8]. Game builds on this work and automates the transformation, using a type and effect system to apply it selectively. A similar idea is also used in [12], which introduces a selective CPS transformation to implement non-local control flow constructs, and is also used to implement delimited continuations in Scala [15]. It has however not been used in the work on defunctionalized interpreters [4], where instead the transformation was applied manually. Furthermore, Game goes beyond the level of the abstract machine, by instantiating it with a supporting runtime, yielding an customized and executable interpreter.

6 Conclusions and Future Work

We presented Game, an environment consisting of a programming language and toolchain for constructing customized interpreters. Using Game, a reusable evaluator is converted into a generic abstract machine, which is subsequently instantiated with a runtime, giving rise to a customized interpreter. Our experiments demonstrate that the high-level SICP evaluator can be reused in three different interpreters, of which the runtime varies in memory management (a non-moving mark-and-sweep versus a moving stop-and-copy GC) and in the stack discipline (managing an explicit stack).

We are currently investigating how to apply Game to other runtime variations, such as tail-call optimization [9], first-class continuations [3], the structure of the interpreter loop [6], and support for implicit parallelization using continuators [10]. The results for GC strengthen our confidence that these concerns can also be injected in high-level evaluators.

References

1. Abelson, H., Sussman, G.J., Sussman, J.: Structure and Interpretation of Computer Programs, 2nd edn. MIT Press/McGraw-Hill, Cambridge (1996)
2. Ager, M.S., Biernacki, D., Danvy, O., Midtgaard, J.: A functional correspondence between evaluators and abstract machines. In: Proc. of the 5th ACM SIGPLAN Intl. Conf. on Principles and Practice of Declaritive Programming, PPDP 2003, pp. 8–19. ACM, New York (2003)
3. Clinger, W.D., Hartheimer, A.H., Ost, E.M.: Implementation strategies for first-class continuations. Higher Order Symbol. Comput. 12, 7–45 (1999)
4. Danvy, O.: Defunctionalized interpreters for programming languages. In: Proc. of the 13th ACM SIGPLAN Intl. Conf. on Functional Programming, ICFP 2008, pp. 131–142. ACM, New York (2008)
5. Diehl, S., Sestoft, P.: Abstract machines for programming language implementation. Future Gener. Comput. Syst. 16, 739–751 (2000)
6. Anton Ertl, M., Gregg, D.: The structure and performance of efficient interpreters. Journal of Instruction-Level Parallelism 5, 1–25 (2003)
7. Ganz, S.E., Friedman, D.P., Wand, M.: Trampolined style. In: Proc. of the Fourth ACM SIGPLAN Intl. Conf. on Functional Programming, ICFP 1999, pp. 18–27. ACM, New York (1999)
8. Graunke, P., Findler, R.B., Krishnamurthi, S., Felleisen, M.: Automatically restructuring programs for the web. In: Proc. of the 16th IEEE Intl. Conf. on Automated Software Engineering, ASE 2001, p. 211. IEEE Computer Society, Washington, DC (2001)
9. Hanson, C.: Efficient stack allocation for tail-recursive languages. In: Proc. of the 1990 ACM Conf. on LISP and Functional Programming, LFP 1990, pp. 106–118. ACM, New York (1990)
10. Herzeel, C., Costanza, P.: Dynamic parallelization of recursive code: part 1: managing control flow interactions with the continuator. In: Proc. of the ACM Intl. Conf. on Object Oriented Programming Systems Languages and Applications, OOPSLA 2010, pp. 377–396. ACM, New York (2010)
11. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: Proc. of the 22nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 1995, pp. 333–343. ACM, New York (1995)
12. Nielsen, L.R.: A selective cps transformation. Electr. Notes Theor. Comput. Sci. 45, 311–331 (2001)
13. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proc. of the ACM Annual Conf., ACM 1972, vol. 2, pp. 717–740. ACM, New York (1972)
14. Rigo, A., Pedroni, S.: Pypy's approach to virtual machine construction. In: Companion to the 21st ACM SIGPLAN Symp. on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 2006, pp. 944–953. ACM, New York (2006)

15. Rompf, T., Maier, I., Odersky, M.: Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. In: Proc. of the 14th ACM SIGPLAN Intl. Conf. on Functional Programming, ICFP 2009, pp. 317–328. ACM, New York (2009)
16. Schrijvers, T., Peyton Jones, S., Chakravarty, M., Sulzmann, M.: Type checking with open type functions. In: Proc. of the 13th ACM SIGPLAN Intl. Conf. on Functional Programming, ICFP 2008, pp. 51–62. ACM, New York (2008)
17. Snyder, M., Frisby, N., Kimmell, G., Alexander, P.: Writing Composable Software with InterpreterLib. In: Bergel, A., Fabry, J. (eds.) SC 2009. LNCS, vol. 5634, pp. 160–176. Springer, Heidelberg (2009)
18. Talpin, J.-P., Jouvelot, P.: The type and effect discipline. Inf. Comput. 111, 245–296 (1994)

Revising and Extending the Uppaal Communication Mechanism

Abdeldjalil Boudjadar, Jean-Paul Bodeveix, and Mamoun Filali

IRIT, University of Toulouse, France
firstname.familyname@irit.fr

Abstract. We study the specification and verification of real-time systems. To deal with the properties of such systems, different toolboxes regarding timing aspects and their related decidable properties have been elaborated: Uppaal [16], Tina [7], Cadp [11] and Kronos [21]. They enable the specification of real-time systems using different formalisms (timed automata and time Petri nets) and the verification of properties expressed in LTL, CTL, TCTL, etc. In this paper we are interested in the Uppaal specification language for which we propose a revised definition of timed automata composition. Moreover, in order to make Uppaal timed automata more expressive, we define an extension enabling instantaneous conditional data communications by superposing message exchange to synchronization. We define an extended timed transition system (ETTS) as a semantic model for both Uppaal timed automata (TA) and Uppaal extended ones for which we have established a compositionality result. In order to reuse the basic Uppaal tool, we give translation rules for transforming a description with the new semantics into a description with the basic Uppaal semantics. Furthermore, to prove the translation correctness, we study the bisimilarity between the ETTS semantics of extended Uppaal TA composition and that corresponding to the translation to basic Uppaal ones.

Keywords: Real-time systems, timed transition systems, verification, composition, timed automata.

1 Introduction

Modern communicating systems are often designed through big assemblies of different components. Because of the interaction and timing constraints of such components, the specification and verification of these systems become a hard task. To dual with such a complexity, expressive specification languages, powerful analysis techniques and associated toolboxes have been proposed and built. In this paper, we are interested in specifications in terms of networks of timed automata [1], their analysis in a hierarchical and compositional way [6] and the associated widely used toolbox: Uppaal [16]. Uppaal is an integrated tool for the modeling, simulation and verification of real-time systems where the properties to be checked are expressed in restricted TCTL [4]. In this paper, we are interested in the Uppaal specification language where we revisit the semantics of Uppaal basic component's (TA) composition so that the semantics of a network of

T. Gschwind et al. (Eds.): SC 2012, LNCS 7306, pp. 114–131, 2012.

components is a product of the semantics of individual components. This makes possible compositional verification [9] which reduces the problem of checking whether a system $S = S_1 \parallel \ldots \parallel S_n$ satisfies a property P to a number of simpler problems of checking whether each component S_i satisfies a property P_i where P is a composition of P_i. It is important to note that S is not actually constructed. Actually:

1. we propose a revised composition with a compositional semantics for Uppaal TA;
2. we define a new extension of Uppaal TA called synchronization with *conditional data communications* which consists in superposing message exchange to synchronization;
3. in order to reuse the basic Uppaal tool, we elaborate translation rules for transforming an extended description into a description with the basic Uppaal semantics.

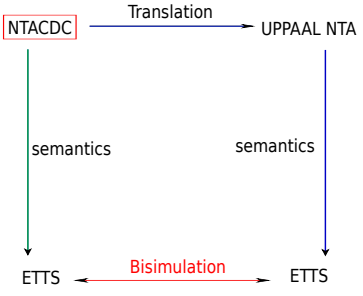


Fig. 1. Extensions Correctness

With respect to our previous paper [8], where we have already presented the composition. In this paper, we go further by specifying the translation rules and establishing their correctness.

The rest of the paper is organized as follows: Section 2 cites the existing related work and presents the motivation and the benefits of our proposal. In section 3, we define the formal basis of our work where we introduce Labelled Transition Systems (LTS) and their bisimilarity, Extended Timed Transition Systems (ETTS), as a low-level timed transition systems, with the corresponding semantics and refinement and establish an associative product of ETTSs. Moreover, we give the semantics of Uppaal timed automata (TA) and networks of timed automata (NTA) in terms of ETTS. In section 4, we define a compositional semantics for the Uppaal TA composition and establish a correctness theorem of the translation of our semantics to that of basic Uppaal. Besides, we define TA with conditional data communications (TACDC) as an extension of Uppaal TA and give both the ETTS and the NTA corresponding to a network of TACDC and study the bisimilarity of the resulting semantics.

2 Related Work

Timed automata[1] have been studied for a long time and different concepts have been introduced like variables, priorities and synchronization to deal with the different aspects of modern complex systems and the underlying composition [5,15,6,10,14,22]. However, a compositional framework for the composition of such structures is lacking. With respect to this issue, we are only

interested in work [5,6] focusing on the composition of extended timed systems with synchronization, committedness and variables.

The authors of [5] define a framework with a non compositional semantics for UPPAAL timed automata product where ,on a synchronization, such a model checks the guards of the involved transitions before applying any action. On the other hand, [6] describes a framework for compositional abstraction and defines a parallel composition operator for UPPAAL timed automata. To establish their results, the authors of [6] restrict both TA and their semantics where both input transition guards (A) and location invariant (B) do not refer to shared variables. Besides, in TA a committed location should have an outgoing transition (C).

Furthermore, in the major part of work[5,6,15,10,14,22], the communication of components is performed via read/write actions on shared variables. Thus, with such a mechanism the modeling of resource allocators and schedulers (with conditional communication) is a difficult task. In this paper, we propose a new semantics for TA composition where the input transition guard is checked after the execution of the output transition action. These changes solve the restrictions (A) and (C). Moreover, in order to make communications easier, we extend TA with instantaneous communication where, transfer of data can happen upon successful synchronization.

Motivating Example. To discuss the benefits of our proposal, let us consider the resource allocator system depicted in Figure 2. According to its requirement, the process *client* specifies the amount of required resources through the assignment of its local variable `need` to the shared variable `req`. Such an amount has to be known by the process *allocator*. Moreover, the variable `resource` represents the quantity of remaining resources of the process *allocator*. The synchronization of both processes *client* and *allocator* takes place through the channel `alloc`. Logically, the *allocator* should grant a request once the required resources are available. Otherwise, the request is postponed. To sum up, the resource allocation transition of the *allocator* is guarded by `req`$\leq$ `resource`. One may remark here that the basic UPPAAL synchronization protocol does not fit our needs because, in UPPAAL, the guards of both synchronizing transitions (sender and receiver) are evaluated *before* the synchronization starts, i.e., in our case, before the process *client* updates the shared variable *req*. We propose a way to trigger such a synchronization where the guard of the receiver (*allocator*) is evaluated after the execution of the update action $req := need$ of the sender (*client*). With respect to our example, this means that the *client*

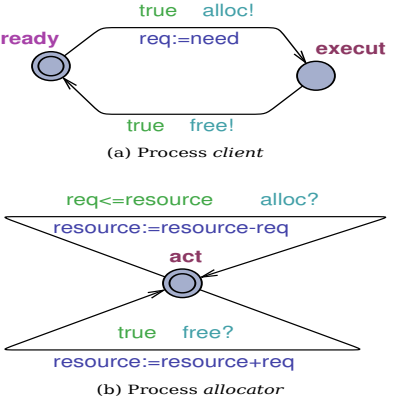


Fig. 2. Resource allocation

first updates the variable shared `req` through the assignment $req := need$, then, the *allocator* grants the request provided its guard evaluates to `true`[1].

3 Transition System Extensions

In this section we briefly introduce one of the fundamental models of concurrency, *transition systems*, originally introduced in [19,3], and since then studied extensively by [18] and others. In fact, transition systems are an elegant model for representing the behavioral aspects. They are essentially composed of states and transitions. States correspond to the configurations reached by the modeled system whereas transitions link these states through the actions of the system. In order to enhance their expressiveness, we consider an extension of classical transition systems where we introduce variables, communications and priority. Due to their safety properties verifiable using model-checking, transition systems have been intensively applied to the modeling of complex systems as well as for giving semantics to synchronous languages and real-time formalisms.

3.1 Labelled Transition Systems

Labelled transition systems [3] are the reference model used to express and to compare behaviors through simulations. They offer a strong notion of equivalence that can be checked efficiently. First, let us start with a brief recall of classical labelled transition systems (LTS) and their bisimulation relation. Throughout this paper, labelled transition systems constitute the lowest semantical level of our study.

Definition 1 (LTS). *A labelled transition system (LTS) over an alphabet Σ is a tuple $\langle Q, Q^0 \subseteq Q, \rightarrow \rangle$ where Q is the state space, $Q^0 \subseteq Q$ is the set of initial states and $\rightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation.*

Here and elsewhere, we note $t : q \xrightarrow{\sigma} q'$ for $(q, \sigma, q') \in \rightarrow$. Moreover, if not needed, the name t of transitions can be omitted. In order to compare LTS's behavior and ensure that a given concrete LTS implements an abstract one, we define LTS simulation.

Definition 2 (Simulation). *Given two LTSs $\mathcal{T}_c = \langle Q_c, Q_c^0, \rightarrow_c \rangle$ (concrete) and $\mathcal{T}_a = \langle Q_a, Q_a^0, \rightarrow_a \rangle$ (abstract), $\mathcal{T}_c$ simulates $\mathcal{T}_a$, we write $\mathcal{T}_c \sqsubseteq_R \mathcal{T}_a$, through a relation $R \subseteq Q_c \times Q_a$ if $\forall q_c \in Q_c^0$, there exists $q_a \in Q_a^0$ such that $R(q_c, q_a)$ and $\forall q_c\ q_c'\ q_a\ \sigma$, if $q_c \xrightarrow{\sigma}_c q_c'$ and $R(q_c, q_a)$ then there exists $q_a' \in Q_a$ such that $q_a \xrightarrow{\sigma}_a q_a'$ and $R(q_c', q_a')$.*

Roughly speaking, two LTSs $\mathcal{T}_i$ and $\mathcal{T}_j$ are bisimilar through a relation R, we write $\mathcal{T}_i \sim_R \mathcal{T}_j$, if $\mathcal{T}_i \sqsubseteq_R \mathcal{T}_j$ and $\mathcal{T}_j \sqsubseteq_{R^{-1}} \mathcal{T}_i$.

[1] The synchronization protocol is atomic: if on the receiver side, the guard does not evaluate to true, updates on the sender side do not take place.

3.2 Timed Transition Systems Extensions

Timed Transition Systems (TTS) [12] are the reference model to define the semantics of real-time formalisms. Basically, a TTS is an LTS where labels can be events or durations. In this section, we define Extended Timed Transition Systems (ETTS) as an extension of TTS by introducing local variables, global variables, static priority (committedness) and location invariants. Moreover, to make transition systems communicate, we specialize the state space and the alphabet to allow several communication protocols:

- via a shared space: we distinguish local and global state spaces updated via a set of actions. These actions can be non-deterministic and blocking.
- via CCS-like channels: we introduce a set C of send-receive channels where two transitions synchronize if their actions are complementary. The resulting transition of such a synchronization corresponds to an internal transition in the composition.
- via CSP-like synchronization: we introduce a set Δ of many-to-many synchronization events. Such a synchronization is used to model a system transition where all of the processes make a lock-step [13]. Throughout this paper, we use this synchronization to model the evolution of time.

Furthermore, *Committedness* is a high level mechanism to express that committed transitions have priority over non-committed ones. Committed transitions starting from a given location hide non committed ones starting from that location. In fact, this notion is relevant when considering composition. Note that hiding is supposed to be static: a non firable committed transition can hide a firable non committed transition. A location q is said to be committed ($\mathrm{Comm}(q) = \top$) if at least one of its outgoing transitions is committed. We have also introduced location invariants to ensure liveness.

Definition 3 (Timed space). *Given the time structure $\langle \Delta, 0, +, \leq \rangle$ where $+$ is associative and commutative, 0 is neutral and $a \leq b \triangleq \exists c,\ a + c = b$, a timed space is a set E together with an operator $\oplus : E \times \Delta \to E$ defining the advance of time. $\oplus$ is supposed to be compatible with the time structure:*

- *Additivity :* $(x \oplus \delta_1) \oplus \delta_2 = x \oplus (\delta_1 \oplus \delta_2)$.
- *Zero-delay :* $x \oplus 0 = x$.

We note $\oplus_\delta$ the function $x \mapsto x \oplus \delta$. In fact, the timed space enables to quantify time and specify its progress (timeliness).

Definition 4 (ETTS). *An Extended Timed Transition System over a shared timed space $\mathcal{G}$ and a set of channels C is a tuple $\langle Q, q^0, \mathcal{L}, \mathcal{I}, I, T, \alpha, \beta, d \rangle$*[2] *where:*

- *Q is the set of locations (local states), q^0 is the initial location,*
- *$\mathcal{L}$ is the local timed space,*
- *$\mathcal{I} \subseteq \mathcal{L} \times \mathcal{G}$ defines the initial states,*

[2] In the rest of the paper, we often omit the projection functions α, β and d.

- $I : Q \to 2^{\mathcal{L}\times\mathcal{G}}$ *associates an invariant to each location,*
- T *is the set of transitions,*
- $\alpha : T \to Q$ *and* $\beta : T \to Q$ *are functions associating respectively the corresponding source and target locations of each transition,*
- $d : T \to 2^{\mathcal{L}\times\mathcal{G}} \times \Lambda \times (\mathcal{L}\times\mathcal{G} \to \mathcal{L}\times\mathcal{G}) \times \mathbb{B}$ *is a function associating to each transition the corresponding guard, label, action (supposed to be deterministic) and committedness flag where* $\Lambda = C? \cup C! \cup \Delta \cup \{\tau\}$ *is the set of labels.* Δ *is the time domain,* $C!$ *and* $C?$ *correspond respectively to send and receive labels over channels of* C*. The action associated to a transition labelled by* $\delta \in \Delta$ *is* $\oplus_\delta \otimes \oplus_\delta$ [3] *which adds* δ *to both local and global parts of the state.*

Furthermore, an ETTS must satisfy a wellformedness condition : time synchronization transitions (with labels in Δ*) are supposed to be non committed.*

We write $t : q \xrightarrow{G/\lambda/a}_b q'$ for $t \in T$ with $\alpha(t) = q, \beta(t) = q', d(t) = \langle G, \lambda, a, b\rangle$. If absent, b is considered to be false. If not needed, the name of a transition will be omitted. Again, the set of transition T is often denoted by the transition relation $\to$. By now, we define formally the predicate $Comm$ by:

$$Comm(q) = \begin{cases} \top & \text{If } \exists\ G\ \lambda\ a\ q' \mid q \xrightarrow{G/\lambda/a}_\top q' \\ \bot & \text{Otherwise} \end{cases}$$

The semantics of an ETTS is specified by its associated LTS defined below. It allows comparing ETTSs through simulation and bisimulation.

Definition 5 (Semantics of an ETTS). *Given the global timed space* $\mathcal{G}$*, the semantics of the ETTS* $\langle Q, q^0, \mathcal{L}, \mathcal{I}, I, T\rangle$ *is the LTS* $\langle Q \times \mathcal{L} \times \mathcal{G}, \{q^0\} \times (\mathcal{I} \cap I(q_0)), \{(q,l,g), \lambda, (q',l',g') \mid (l,g) \models I(q),\ (l',g') \models I(q'),\ \exists G, \exists\ a, \exists\ b, \exists t : q \xrightarrow{G/\lambda/a}_b q'\ \in T,\ (l,g) \in G,\ \text{and } ((l,g),(l',g')) \in a \text{ and } \neg b \Rightarrow q \nrightarrow_\top\}\rangle$.

The notation $q \nrightarrow_\top$ states the absence of outgoing committed transitions from q. Moreover, we state by $(l,g) \in G$ that the valuation of both local and global spaces satisfies the guard G. Here, one can remark that the ETTS non-committed transitions outgoing from a committed location are not held in the corresponding LTS semantics because they are hidden. In fact, only non-hidden enabled ETTS transitions are held. In the meantime, we define the similarity of ETTSs through their LTSs semantics.

Definition 6 (Similarity). *An ETTS* $\mathcal{T}_i$ *is said to be (bi)similar to an ETTS* $\mathcal{T}_j$ *if their associated LTSs are (bi)similar.*

In fact, such a definition enables to get rid managing priorities during bisimulations proofs.

[3] The function composition operator $\otimes$ is defined by: $(f \otimes g)(x,y) = (f(x), g(y))$.

Restriction of an ETTS. On composition of ETTSs, unmatched synchronizing transitions are ignored. We define the corresponding operation so-called restriction [17]. In fact, the restriction of an ETTS over a set of channels is an ETTS where transitions composable over these channels have been deleted.

Definition 7 (ETTSs restriction). *Let $\mathcal{T} = \langle Q, q_0, \mathcal{L}, \mathcal{I}, I, \longrightarrow \rangle$ be an ETTS over a set of channels C. Let $C' \subseteq C$, we define the restriction of $\mathcal{T}$ to C', denoted $\mathcal{T}\backslash C'$, to be the ETTS $\langle Q, q_0, \mathcal{L}, \mathcal{I}, I, \longrightarrow \backslash \{q \xrightarrow{G/\lambda/a}_b q' \mid \lambda \in C'! \cup C'?\}\rangle$.*

3.3 Product of ETTSs

In this section, we define an associative n-ary product of ETTSs where the local space of composition is simply obtained by the merge of individual local spaces. In a simpler case, we may consider the merge as a renaming of the corresponding variables. Moreover, guards, actions and location invariants of each individual ETTS, depending on its local space, are extended according to the new local space of the composition.

Definition 8 (N-ary product of a family of ETTSs). *Given an indexed family of n ETTSs $\mathcal{T}_i = \langle Q_i, q_i^0, \mathcal{L}_i, \mathcal{I}_i, I_i, \rightarrow_i \rangle$ over the same shared space $\mathcal{G}$, their product $\Pi_i \mathcal{T}_i$ is defined by the ETTS over $\mathcal{G}$:*

$$\langle \times_i Q_i, \langle q_1^0, .., q_n^0 \rangle, \times_i \mathcal{L}_i, \cap_i \mathcal{I}_i \uparrow_i, \langle q_1, .., q_n \rangle \mapsto \cap_i I_i(q_i) \uparrow_i, \quad \rightarrow \rangle$$

where for $E \subseteq \mathcal{L}_i \times \mathcal{G}$, its extension to $(\times_i \mathcal{L}_i) \times \mathcal{G}$ is defined by: $E \uparrow_i \triangleq \{\langle l, g\rangle \mid \langle l_i, g\rangle \in E\}$, for $a : \mathcal{L}_i \times \mathcal{G} \rightarrow \mathcal{L}_i \times \mathcal{G}$ its extension to $(\times_i \mathcal{L}_i) \times \mathcal{G} \rightarrow (\times_i \mathcal{L}_i) \times \mathcal{G}$ is defined by: $a \uparrow_i \langle l, g\rangle \triangleq \langle l[i \mapsto a(\langle l_i, g\rangle) \downarrow_1], a(\langle l_i, g\rangle) \downarrow_2\rangle$. $\rightarrow$ is the smallest relation such that:

$$\frac{t_i : q_i \xrightarrow{G/\lambda/a}_{i,b} q_i' \quad \lambda \in C! \cup C?}{q \xrightarrow{G\uparrow_i/\lambda/a\uparrow_i}_b q[i \leftarrow q_i']} \; ASYNC(t_i)$$

$$\frac{(\forall i)\ q_i \xrightarrow{G_i/\delta/\oplus_\delta \otimes \oplus_\delta}_{i,\perp} q_i' \ \delta \in \Delta}{q \xrightarrow{\wedge_i G_i\uparrow_i/\delta/(\otimes_i \oplus_\delta)\otimes\oplus_\delta}_\perp q'} \; TIME$$

$$\frac{t_i : q_i \xrightarrow{G/\tau/a}_{i,b} q_i' \quad (\bigvee_j Comm(q_j)) \Rightarrow b}{q \xrightarrow{G\uparrow_i/\tau/a\uparrow_i}_b q[i \leftarrow q_i']} \; TAU(t_i)$$

$$\frac{t_i : q_i \xrightarrow{G_i/c!/a_i}_{i,b_i} q_i' \qquad t_j : q_j \xrightarrow{G_j/c?/a_j}_{j,b_j} q_j' \qquad i \neq j \quad (\bigvee_k Comm(q_k)) \Rightarrow (b_i \vee b_j)}{q \xrightarrow{G_i\uparrow_i \wedge ((G_j\uparrow_j)\circ(a_i\uparrow_i))/\tau/a_j\uparrow_j \circ a_i\uparrow_i}_{b_i \vee b_j} q[i \leftarrow q_i', j \leftarrow q_j']} \; SR(t_i, t_j)$$

The locations, respectively initial locations, of the product are obtained by merging the locations, respectively initial, locations, of individual ETTS. Moreover, the initial valuations $\mathcal{I}$, respectively invariant I, of the composition are defined

through the intersection of initial valuations, respectively invariants, of individual $\mathcal{T}_i$. The notation $q[i \leftarrow q'_i]$ states the replacement of the ith location of state q by location q'_i. Here, one may distinguish that, on a synchronization, the input transition guard is checked after the execution of the output transition action. Moreover, $Comm(q) \Rightarrow b$ with $t : q_i \xrightarrow{G/\lambda/a}_{i\ b} q'_i$ specifies that t must be committed if there exists another outgoing committed transition from q. Otherwise stated: a prioritized transition cannot be hidden by a less prioritized transition. In fact, through rule $ASYNC(t_i)$, we allow the ETTS $\mathcal{T}_i$ to hold its synchronizing transition t_i, for a future composition, without checking its committedness flag because it may be that a compatible committed transition will synchronize with t_i, making then the resulting transition committed. The rule $TIME$ states that a delay δ of the composition may occur when each component performs a delay δ. In fact, through $\oplus_\delta \otimes \oplus_\delta$ we specify the progress of both local and global clock variables. $TAU(t_i)$ induces an internal transition of the composition from the transition t_i of component $\mathcal{T}_i$. Through such a rule, the corresponding transition of the composition is non-committed if each location q_i of the current state is non-committed. Otherwise, $TAU(t_i)$ provides that t_i has priority over the other transitions outgoing from q. Finally, $SR(t_i, t_j)$ (for send/receive) describes the synchronization of components $\mathcal{T}_i$ and $\mathcal{T}_j$ on channels $c \in C$ where the guard G_j of the receiver is checked after simulating the update made by the sender action a_i. In fact, two components synchronize if their synchronizing transitions are compatible. The resulting transition, labelled by the internal event τ, is committed if either the *send* or the *receive* transition is committed. Otherwise, a non-committed synchronization may only occur if the current state q is non-committed.

Theorem 1 (Generalized associativity). *The product of ETTSs is associative, i.e.:* $\Pi_{i\in\mathcal{I}}(\Pi_{j\in\mathcal{J}_i}\mathcal{T}_{i,j}) \sim \Pi_{i\in\mathcal{I},j\in\mathcal{J}_i}\mathcal{T}_{i,j}$.

Proof. It essentially consists in defining an isomorphism between the two structures (state space and transitions) preserving labels and priorities.

3.4 Uppaal Networks of Timed Automata

In this section, we consider Uppaal timed automata (TA) [5,6] and networks of timed automata (NTA). In fact, they are an extension of the classical ones where local variables, global variables and other high level concepts like committedness and communication have been introduced. Moreover, location committedness is a static priority relation where transition enabledness is not taken into account. It defines two priority levels where transitions outgoing from committed locations have priority over others.

Notations. Given a set V of variables valued in a domain D.

- $\mathfrak{E}(V)$ defines the set of expressions built over V.
- $\mathfrak{P}(V)$ defines the set of predicates built over V.

- $\uplus_i V_i \triangleq \cup_i \{i\} \times V_i$.
- $\uparrow_i$ is a function converting each expression of $\mathfrak{E}(V_i)$ to an expression of $\mathfrak{E}(\uplus_j V_j)$ by renaming variables $v \in V_i$ into $(j, v) \in \uplus_j V_j$.

According to former notations, we introduce the following semantics functions:

- $p \in \mathfrak{P}(V)$, $[\![p]\!] \subseteq V \to D$.
- $e \in \mathfrak{E}(V)$, $[\![e]\!] \in (V \to D) \to D$.
- For an action $a : V \nrightarrow \mathfrak{E}(V)$, usually denoted $v_1 := e_1 \parallel \ldots \parallel v_n := e_n$ where $\{v_1 \ldots v_n\} = dom(a)$ and $e_i = a(v_i)$, its semantics $[\![a]\!] : (V \to D) \to (V \to D)$ is defined by $env \mapsto (v \mapsto \begin{cases} [\![a(v)]\!]_{env} & \text{if } v \in dom(a) \\ env(v) & \text{otherwise} \end{cases})$
- The semantics of a sequence of actions is defined by: $[\![a_i; a_j]\!] \triangleq [\![a_j]\!] \circ [\![a_i]\!]$.
- For a predicate P with $[\![P]\!] \to 2^{V \to D}$, we outline the following semantics:
  $[\![P(\_)]\!] : D \to 2^{V \to D}$
  $[\![e]\!] : (V \to D) \to D$
  $[\![P(e)]\!] = \{u \in V \to D \mid u \in [\![P(\_)]\!]([\![e]\!](u))\}$

Note that the restrictions of both expressions and predicates construction are not detailed here. By now, we give the Uppaal semantics of both TA and networks of TA through extended timed transition systems (ETTSs).

Timed Automata. Timed automata has been introduced as a modeling framework, a basic mathematical framework to support the description and analysis of systems. In such a formalism, a process is modeled as an automaton. An automaton is structured as a set of states linked through a number of transitions, going from state to state, denoting the execution of elementary actions, the basic unit of behavior. Also, there is an initial state (sometimes, more than one) and eventual final states. Structurally, a timed automaton [2,1] is a finite-state machine extended with a finite collection of real-valued clocks initialized to zero and increased synchronously. In this section, we consider the set of variables split to local and global ones. This distinction has been established to make TA structure composable according to the Uppaal NTA definition.

Definition 9 (Timed automaton). *A timed automaton on a set of clocks χ, a set of global variables $\mathcal{V}^g$, its initialization function $Init^g \in \mathcal{V}^g \to D$ and a set of channels C is a tuple $\langle Q, q^0, K, \mathcal{V}^l, Init, I, T, \alpha, \beta, Grd, Act, s\rangle$ [4] where:*

- *Q is a set of locations, $q^0 \in Q$ is the initial location,*
- *$K \subseteq Q$ is a set of committed locations,*
- *$\mathcal{V}^l$ is a set of local variables,*
- *$Init^l : \mathcal{V}^l \to D$ is the initialization function of local variables,*
- *$I : Q \to \mathfrak{P}(\mathcal{V}^l \cup \mathcal{V}^g \cup \chi)$ associates an invariant to each location,*
- *T is the set of transitions,*
- *$\alpha : T \to Q$ and $\beta : T \to Q$ are functions associating respectively the corresponding source and target locations of each transition,*

[4] We assume that $\chi \cap \mathcal{V}^g = \mathcal{V}^l \cap \mathcal{V}^g = \mathcal{V}^l \cap \chi = \emptyset$.

- $Grd : T \to \mathfrak{P}(\mathcal{V}^l \cup \mathcal{V}^g \cup \chi)$ *is a function associating to each transition the corresponding guard,*
- $Act : T \to ((\mathcal{V}^l \cup \mathcal{V}^g \cup \chi) \nrightarrow \mathfrak{E}(\mathcal{V}^l \cup \mathcal{V}^g))$ *is a partial function associating to each transition the corresponding action. In fact, actions assign to a subset of variables of* $\mathcal{V}^l \cup \mathcal{V}^g$ *a formula built on variables of* $\mathcal{V}^l \cup \mathcal{V}^g$*,*
- $s : T \to C? \cup C! \cup \{\tau\}$ *is a function associating to each transition a label.*

Here and elsewhere, we note $t : q \xrightarrow{G/\lambda/a} q'$ for $t \in T$ with $\alpha(t) = q, \beta(t) = q', Grd(t) = G,\ Act(t) = a,\ s(t) = \lambda$. As previously stated, we often omit T and the projection functions $\alpha,\ \beta,\ Grd,\ Act$ and s. They are implicitly given by the transition relation $\to$. In order to study the bisimulation of timed automata, different semantics of TA through TTS have been proposed [6,15,14]. Here, we define the semantics of TA over ETTSs. First, let us specify the timed space $(\mathcal{G}, \oplus)$ associated to a timed automaton by the following:

- $\mathcal{G} = (\mathcal{V}^g \to D) \times (\chi \to \mathbb{R}^+)$
- $(u^g, u^\chi) \oplus \delta = (u^g, x \mapsto u^\chi(x) + \delta)$

Definition 10 (ETTS of a TA). *Given a set of channels* C*, a set of global variables* $\mathcal{V}^g$ *with its initialization function* $Init^g$ *and a set* χ *of clocks. The semantics of a timed automaton* $\langle Q, q^0, K, \mathcal{V}^l, Init^l, I, \to_{ta}\rangle$ *is defined by the ETTS* $\langle Q, q^0, (\mathcal{V}^l \to D), \mathcal{I}, I, \to\rangle$ *over the timed space* $(\mathcal{G}, \oplus)$ *where* $\mathcal{I} = \{(Init^l, (Init^g, \chi \times \{0\}))\}$ *and* $\to$ *is the smallest relation such that:*

$$\frac{q \xrightarrow{G/\lambda/a}_{ta} q'}{q \xrightarrow{[\![G]\!]/\lambda/[\![a]\!]}_{q \in K} q'}\ Action \qquad \frac{q \in K}{q \xrightarrow{\bot/\tau/\mathrm{Id}_{\mathcal{L}\times\mathcal{G}}}_{\top} q}\ Empty$$

$$\frac{(q \notin K)}{q \xrightarrow{\top/\delta/Id_{\mathcal{L}}\ \otimes\ \oplus_\delta}_{\bot} q}\ Delay$$

The rule *Action* associates to each TA transition an ETTS transition labelled by the same event, guarded by the semantics of the TA transition guard and engaging with the semantics of the corresponding TA action. The *Empty* transition (identity) is not firable, it is especially used to hold the committedness information of a TA location when the former does not have any outgoing transition. Furthermore, from each TA non-committed location, the *Delay* rule specifies a non-committed ETTS transition outgoing from that location. Such a transition updates only clock (global) variables. One can remark that both *Empty* and *Delay*-transitions do not modify local states (locations).

Simulation. We say that a TA $\mathcal{T}_c$ refines another TA $\mathcal{T}_a$ if the simulation relation holds between their associated ETTSs: $\mathcal{T}_c \sqsubseteq \mathcal{T}_a \triangleq ETTS(\mathcal{T}_c) \sqsubseteq ETTS(\mathcal{T}_a)$.

Composition of Timed Automata. To model compound systems wherein several components are concurrently run, timed automata are concurrently composed giving rise to networks of timed automata. Every automaton may fire a

transition separately. The state of the system (network) is then described by the vector of the locations reached in automata. The Uppaal language [16] has adopted the CCS parallel composition semantics where, according to [6], the NTA semantics[5] is not compositional.

Definition 11 (Network of timed automata). *A network of timed automata on a set of shared variables $\mathcal{V}^g$ is a finite collection of timed automata defined on the same set of clocks χ and channels C.*

In order to study the properties of timed automata composition, such as compositionality, we recall the semantics of Uppaal NTA according to [5] where, on a synchronization, Uppaal checks that both involved transition's guards are simultaneously satisfied.

Definition 12 (NTA semantics). *Given a network of timed automata $\langle Q_i, q_i^0, K_i, \mathcal{V}_i^l, Init_i^l, I_i, \rightarrow_i \rangle_{i\in\{1..n\}}$ defined on the same set of channels C, clocks χ and global variables $\mathcal{V}^g$ with the initialization function $Init^g$, its semantics is defined by the ETTS $\langle \times_i Q_i, \langle q_1^0, \ldots, q_n^0 \rangle, (\biguplus_i \mathcal{V}_i^l) \rightarrow D, \mathcal{I}, I, \rightarrow \ \rangle$ over the timed space $(\mathcal{G}, \oplus)$ where $\mathcal{I} = \{((i,v) \mapsto Init_i^l(v), (Init^g, \chi \times \{0\}))\}$, $I(q) = \bigwedge_i I_i(q_i) \uparrow_i$ and $\rightarrow$ is the smallest relation such that:*

$$\frac{q_i \xrightarrow{G/\tau/a}_i q_i' \quad (\bigvee_j q_j \in K_j) \Rightarrow q_i \in K_i}{q \xrightarrow[q_i\in K_i]{[\![G\uparrow_i]\!]/\tau/[\![a\uparrow_i]\!]} q[i \leftarrow q_i']}\ Tau_i \qquad \frac{\bigwedge_i q_i \notin K_i}{q \xrightarrow{\top/\delta/Id_{\mathcal{L}} \otimes \oplus_\delta}_\perp q}\ Delay$$

$$\frac{q_i \xrightarrow{G_i/c!/a_i}_i q_i' \quad q_j \xrightarrow{G_j/c?/a_j}_j q_j' \quad i \neq j \quad (\bigvee_k q_k \in K_k) \Rightarrow q_i \in K_i \vee q_j \in K_j}{q \xrightarrow[q_i\in K_i \vee q_j \in K_j]{[\![G_i\uparrow_i]\!]\wedge[\![G_j\uparrow_j]\!]/\tau/[\![a_j\uparrow_j]\!]\circ[\![a_i\uparrow_i]\!]} q[i \leftarrow q_i', j \leftarrow q_j']}\ SR_{i,j}$$

Due to the extension of each TA local variables $\mathcal{V}_i^l$ to $\biguplus_i \mathcal{V}_i^l$, both guards, actions and location invariants of each individual TA are syntactically extended to the new set of local variables. The guards and actions of the underlying ETTS are defined by the semantics of TA extended guards and actions. The rule *Delay* is earlier presented in the semantics of a TA. Moreover, rules TAU_i and $SR_{i,j}$ are respectively similar to rules TAU and SR of the ETTS product. We may remark that only non-hidden transitions of NTA are held in the ETTS semantics.

4 Extensions of Uppaal

In this section, we propose a revised definition of timed automata composition with a compositional semantics. Moreover, in order to enhance Uppaal timed automata, we define an extension enabling instantaneous conditional data communications by superposing message exchange to synchronization. For each extension, a translation to basic Uppaal is established.

4.1 Compositional Uppaal

Similarly to the product of ETTSs, we define a new timed automata composition where the input transition guard is only checked after taking into account the effect of the corresponding output transition action.

Definition 13 (Compositional NTA semantics). *Given a network of timed automata* $\langle Q_i, q_i^0, K_i, \mathcal{V}_i^l, Init_i, I_i, \rightarrow_i \rangle_{i\in\{1..n\}}$ *defined on the same set of channels* C*, clocks* χ *and global variables* $\mathcal{V}^g$*, its compositional semantics is defined by the ETTS* $\langle \times_i Q_i, \langle q_1^0, \ldots, q_n^0 \rangle, (\biguplus_i \mathcal{V}_i^l) \rightarrow D, \mathcal{I}_i, I, \rightarrow \ \rangle$ *given by Definition 12 where the transition relation* $\rightarrow$ *is redefined by rules* TAU_i*, Delay and the following rules:*

$$\frac{\begin{array}{c} q_i \xrightarrow{G_i/c!/a_i}_i q_i' \quad q_j \xrightarrow{G_j/c?/a_j}_j q_j' \quad i \neq j \\ b = (q_i \in K_i \vee q_j \in K_j) \quad , \quad (\bigvee_k q_k \in K_k) \Rightarrow b \end{array}}{q \xrightarrow{[\![G_i\uparrow_i]\!]\wedge([\![G_j\uparrow_j]\!]\circ[\![a_i\uparrow_i]\!])/\tau/[\![a_j\uparrow_j]\!]\circ[\![a_i\uparrow_i]\!]}_b \ q[i \leftarrow q_i', j \leftarrow q_j']} SR_{i,j}$$

$$\frac{q_i \in K_i}{q \xrightarrow{\perp/\tau/\mathrm{Id}_{\mathcal{L}\times\mathcal{G}}}_\top q} Empty$$

The rule *Empty* is earlier presented in the semantics of a TA. Hence, on a synchronization $SR_{i,j}$, the input transition guard $G_j \uparrow_j$, extended on the new set of local variables, is checked after the execution of the output transition (extended) action $a_i \uparrow_i$. One can remark that both Uppaal semantics and our compositional one are the same if all of the input transition guards equal to **true**. In the following, the compositionality of such a semantics is established.

Theorem 2 (Compositional semantics). *The ETTS of a network of timed automata is bisimilar to the restriction to time and* τ*-transitions of the product of ETTSs associated to individual TA. Formally,* $ETTS(NTA) \sim \Pi_i ETTS(TA_i)\backslash C$.

Proof. It is direct because we have the same composition rules in both sides. The difference resides in the occurrence of unmatched communication transitions in the ETTSs product, but these transitions will be suppressed when applying the restriction operator. This result is mainly another formulation of the result that we formally proved with the Coq theorem prover [8]. □

4.2 Translation of Compositional Uppaal to Basic Uppaal

In order to reuse the Uppaal tool, we outline a *model transformation* converting a compositional network of timed automata into a new network of timed automata to be analyzed by Uppaal. The proposed transformation is decomposed into three basic transformations where each one preserves the semantics of the transformed NTA. The composition of these transformations applied on the original network of timed automata generates a NTA such that Uppaal semantics and our proposed semantics are the same. It follows that we can check

a NTA, through the Uppaal model checker, according to our semantics. In the following, we outline a *normal form* of networks of timed automata for which Uppaal semantics and our proposed one are bisimilar. Thereafter, we give the basic transformations, we establish their correctness and their composition.

Networks of Timed Automata Normal Form. A network of timed automata is said to be in normal form if:

- Only global data is used.
- each channel is used by at most one sender and one receiver. Such channels are called "one one" channels.
- Channel receptions are not guarded, i.e., the expression used to guard any reception is `true`.

Properties. Given a normalized timed automata, its Uppaal semantics and its semantics according to our proposal are the same.

Translation. The basic idea of transformation is to move the "late" reception guards to the sending point.

Step 1: Variables Globalization. Local variables occurring in guards of receptions become global. This transformation allows to evaluate guards out of their local initial context. Formally, given a NTA $N = \langle \mathcal{V}^g, Init^g, \chi, C, \langle Q_i, q_i^0, K_i, \mathcal{V}_i^l, Init_i^l, I_i, T_i, \alpha_i, \beta_i, Grd_i, Act_i, s_i \rangle_i \rangle$, its transformation through this step is the NTA $Step1(N) = \langle (\biguplus_i \mathcal{V}_i^l) \cup \mathcal{V}^g, Init, \chi, C, \langle Q_i, q_i^0, K_i, \emptyset, Init_i^l, I_i \uparrow_i, T_i, \alpha_i, \beta_i, Grd_i', Act_i', s_i \rangle_i \rangle$ where

$$\begin{cases} Init(\langle i, x \rangle) = Init_i^l(x) & \textbf{if } x \in V_i^l \\ Init(y) = Init^g(y) & \textbf{if } y \in V^g \\ Grd_i' : T_i \to \mathfrak{P}(\biguplus_i \mathcal{V}_i^l \cup \mathcal{V}^g \cup \chi) & \forall t \in T_i,\ Grd_i'(t_i) = Grd_i(t_i) \uparrow_i \\ Act' : T_i \to ((\Gamma \cup \chi \cup \biguplus_i \mathcal{L}_i) \nrightarrow \mathfrak{E}(\Gamma \cup \biguplus_i \mathcal{L}_i)) & \forall t \in T_i,\ Act_i'(t_i) = Act_i(t_i) \uparrow_i \end{cases}$$

Note that both guard and action scopes are extended to the new set of variables.

Theorem 3 (Bisimulation). *An NTA and its translation through Step1 are bisimilar for both Uppaal and compositional semantics.*

Proof. It consists in defining a mapping between original variables and their globalization, which is a renaming.

Step 2: Unique Reception Transition. This transformation allows to retrieve the succeeding receptions. Each reception is distinguished by a dedicated channel. Formally, given a NTA $N = \langle \mathcal{V}^g, Init^g, \chi, C, \langle Q_i, q_i^0, K_i, \mathcal{V}_i^l, Init_i^l, I_i, T_i, \alpha_i, \beta_i, Grd_i, Act_i, s_i \rangle_i \rangle$, its transformation through this step is the NTA $Step2(N) = \langle \mathcal{V}^g, Init^g, \chi, \bigcup_i \{t_i | s_i(t_i) \in C?\}, \langle Q_i, q_i^0, K_i, \mathcal{V}_i^l, Init_i^l, I_i, T', \alpha_i', \beta_i', Grd_i', Act_i', s_i' \rangle_i \rangle$ where:

- $T' = \bigcup_{j,c \in C} \{\langle t_i, t_j \rangle \mid s_i(t_i) = c! \wedge s_j(t_j) = c?\} \cup \{t_i | s_i(t_i) \notin C!\}$
- $\alpha'_i(\langle t_i, t_j \rangle) = \alpha_i(t_i)$, $\alpha'_i(t_i) = \alpha_i(t_i)$
- $\beta'_i(\langle t_i, t_j \rangle) = \beta_i(t_i)$, $\beta'_i(t_i) = \beta_i(t_i)$
- $s'_i(\langle t_i, t_j \rangle) = t_i!$, $s'_i(t_i) = \tau$ if $s_i(t_i) = \tau$ else $t_i?$
- $Grd'_i(\langle t_i, t_j \rangle) = Grd_i(t_i)$, $Grd'_i(t_i) = Grd_i(t_i)$
- $Act'_i(\langle t_i, t_j \rangle) = Act_i(t_i)$, $Act'_i(t_i) = Act_i(t_i)$

Thus, we establish the following theorem:

Theorem 4 (Bisimulation). *The compositional semantics of an NTA and that of its translation through Step2 are bisimilar.*

Proof. When transforming, each input transition is translated with a new name and a new channel for each transition name. Similarly, each output transition t on a channel c will be translated with a new name (t, c) and labelled with the channel of the corresponding translated input. In fact, the proof consists in finding a mapping between output and input transitions where an output transition (t_i, c_i) synchronizes with an input one t_j if $s'_i(t_i) = s'_j(t_j)$ and $s_i(t_i) = s_j(t_j) = c$

Lemma 1 (Reception existence). *In Step*$2(N)$*, for each channel* $c \in C$*, there exists at least one reception transition on* c*, i.e. labeled by* $c?$*. Formally,* $\exists f_i : C \to \bigcup_i \{t \in T_i \mid s_i(t) \in C?\}$

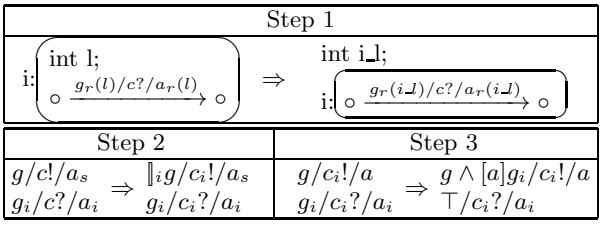
Lemma 2 (Reception uniqueness). *In Step*$2(N)$*, for each channel* $c \in C$*, there exists a unique reception transition on* c*, i.e. labelled by* $c?$*. Formally,* $\forall t \in T_i,\ t' \in T_j,\ c,\ (s_i(t) = c?\ \wedge\ s_j(t') = c?) \Rightarrow (i = j\ \wedge t = t')$.

Step 3: Reception Guards Elimination. The reception transition guard is moved to the sender side after calculating the effect of the sender action. The correctness of this transformation relies on the fact that there is no local variables and for each channel there is an unique reception transition (denoted by the function r). Formally, the transformation of a NTA $\langle \mathcal{V}^g, Init^g, \chi, C, \langle Q_i, q_i^0, K_i, \emptyset, Init_i^l, I_i, T_i, \alpha_i, \beta_i, Grd_i, Act_i, s_i \rangle_i \rangle$ through this step is the NTA where each function Grd_i is replaced by:

$$Grd'_i(t) = \begin{cases} true & \text{if } s_i(t) \in C? \\ Grd_i(t) \wedge [Act_i(t)]Grd_j(t') \ where \langle j, t' \rangle = r(c) & \text{if } s_i(t) = c! \\ Grd_i(t) & \text{Otherwise} \end{cases}$$

Theorem 5 (Bisimulation). *The compositional semantics of an NTA and that of its translation through Step3 are bisimilar.*

Proof. It is straightforward.
We summarize the preceding steps by the following table.



Translation Correctness Theorem. Through the composition of the former basic transformations, we conclude that the semantics of a compositional NTA and that corresponding to its translation to an Uppaal one are similar.

Theorem 6 (Translation Correctness). *Given a NTA N, the compositional semantics of N and the Uppaal one (non compositional) of the translation $Step3 \circ Step2 \circ Step1(N)$ are bisimilar.*

4.3 Compositional Uppaal with Conditional Data Communication

We extend timed automata by instantaneous data communication which consists in superposing message exchange to synchronization. We use the notation $c!e||c?x$ to designate a synchronization on a channel c with the communication of an expression e of the sender. The value of e will be assigned to the local variable x of the receiver. Furthermore, we constrain the reception by an additional condition $c?x$ **where** $P(x)$, expressing that a sender and a receiver can synchronize with a data communication if the communicated data satisfies the predicate P. Otherwise, the synchronization will be blocked, i.e. the communicated data is ignored and the update made by the output transition is not applied.

TA with Conditional Data Communication (TACDC). First, we give a formal definition of TA with conditional data communication where the guard G of each input communicating transition $q \xrightarrow{G/c?x \textbf{ where } P(x)/a} q'$ does not depend on variable x. Moreover, we carry out the translation of TACDC to basic TA.

Definition 14 (TA with conditional data communication). *Given a set of clocks χ, a set of global variables $\mathcal{V}^g$ with its initialization function $Init^g$ and a set of actions Act, a timed automaton with data communication is a tuple $\langle Q, q^0, K, \mathcal{V}^l, Init^l, I, \rightarrow\rangle$ where $\langle Q, q^0, K, \mathcal{V}^l, Init^l, I\rangle$ are defined as in TA and the transition relation $\rightarrow$ is defined by $\rightarrow \subseteq Q \times \mathfrak{P}(\mathcal{V}^l \cup \mathcal{V}^g \cup \chi) \times Act \times \Sigma \times Q$ where the set of labels $\Sigma = \{C? \times \mathcal{V}^l \times \mathfrak{P}(\mathcal{V}^l \cup \mathcal{V}^g)\} \cup \{C! \times \mathfrak{E}(\mathcal{V}^l \cup \mathcal{V}^g)\} \cup \{\tau\}$.*

In order to check the properties of TACDC, we define their translation into compositional Uppaal TA, introduced in *Section 4.1*, where the communication of data is performed through read-write actions on shared variables. Moreover, in such a communication, the received value must satisfy the constraint P.

Definition 15 (TA corresponding to TACDC). *Given a set of channels C, a set of global variables $\mathcal{V}^g$ with its initialization function $Init^g$ and a set $\mathcal{X}$ of clock variables, a timed automaton with conditional data communication $\langle Q, q^0, K, \mathcal{V}^l, Init^l, I, \rightarrow_c\rangle$ is the TA $\langle Q, q^0, K, \mathcal{V}^l, Init^l, I, \rightarrow\rangle$ defined on C, χ and global variables $\mathcal{V}^g \cup \{sh\}$* [5] *where sh is a new shared variable and the transition relation $\rightarrow$ is defined by:*

[5] Practically, *sh* will be typed as the union of all the message types.

$$\frac{q \xrightarrow{G/\tau/a}_c q'}{q \xrightarrow{G/\tau/a} q'}\,Tau \qquad \frac{q \xrightarrow{G/\lambda!e/a}_c q' \quad \lambda \in C}{q \xrightarrow{G/\lambda!/sh:=e;a} q'}\,Send$$

$$\frac{q \xrightarrow{G/\lambda?x \textbf{ where } P(x)/a}_c q' \quad \lambda \in C}{q \xrightarrow{G \wedge P(sh)/\lambda?/x:=sh;a} q'}\,Receiv$$

Through rule Tau, the internal transitions of TACDC are still unchanged. However, in communicating ones ($Send$, resp $Receiv$) the data exchange is carried out through an assignment into, respectively from, the variable sh. Moreover, the predicate P is evaluated on the value of sh when checking the input transition guard.

Networks of TACDC. In this section, we consider the networks (composition) of TACDC and study the bisimilarity between their ETTS (direct) semantics and that corresponding to their translation to Uppaal NTA. In fact, we define the semantics of networks of TACDC through ETTSs where each conditional data communication is performed through the assignment of communicated data to the corresponding receiver local variable. Furthermore, the resulting transition is again guarded by the predicate $P(x)$. According to our composition, the satisfaction of both input transition guard and predicate $P(x)$ is only checked after the execution of the output transition action.

Definition 16 (Semantics of a network of TACDC). *Given a network of TACDC* $\langle Q_i, q_i^0, K_i, \mathcal{V}_i^l, Init_i^l, I_i, \rightarrow_i \rangle_i$ *defined on the same set of channels* C*, clocks* χ *and a set of global variables* $\mathcal{V}^g$ *with the initialization function* $Init^g$*, its semantics is defined by the ETTS* $\langle \times_i Q_i, \langle q_1^0, \ldots, q_n^0 \rangle, (\biguplus_i \mathcal{V}_i^l) \rightarrow D, \mathcal{I}, I, \rightarrow \rangle$ *over the timed space* $(\mathcal{G}, \oplus)$ *and the set of channels* C *where* $\mathcal{I} = \{((i,v) \mapsto Init_i^l(v), (Init^g, \chi \times \{0\}))\}$*,* $I(q) = \bigwedge_i I_i(q_i) \uparrow_i$ *and* $\rightarrow$ *is defined by:*

$$\frac{q_i \xrightarrow{G/\tau/a}_i q_i' \quad (\bigvee_j q_j \in K_j) \Rightarrow q_i \in K_i}{q \xrightarrow[q_i \in K_i]{[\![G\uparrow_i]\!]/\tau/[\![a\uparrow_i]\!]} q[i \leftarrow q_i']}\,TAU_i \qquad \frac{\bigwedge_i q_i \notin K_i}{q \xrightarrow{\top/\delta/Id_{\mathcal{L}} \otimes \oplus_\delta}_\perp q}\,Delay$$

$$\frac{\begin{array}{c} q_i \xrightarrow{G_i/c!e/a_i}_i q_i' \quad q_j \xrightarrow{G_j/c?x \textbf{ where } P(x)/a_j}_j q_j' \\ i \neq j \quad b = (q_i \in K_i \vee q_j \in K_j) \quad (\bigvee_k q_k \in K_k) \Rightarrow b \end{array}}{q \xrightarrow{[\![G_i\uparrow_i]\!] \wedge ((([\![G_j\uparrow_j]\!] \wedge [\![P(\_)\uparrow_j]\!])([\![e\uparrow_i]\!])) \circ ([\![a_i\uparrow_i]\!]))/\tau/[\![a_j\uparrow_j]\!] \circ [\![a_i\uparrow_i]\!] \circ [\![x\uparrow_j := e\uparrow_i]\!]}_b q[i \leftarrow q_i', j \leftarrow q_j']}\,SR_{i,j}$$

Again, both guards and actions of individual TA are extended on the new set of local variables. All of the rules TAU_i, $Delay$ and $SR_{i,j}$ are earlier explained. Moreover, for SR where the communicated data e is assigned to the local variable x of the receiver, one can remark that the Uppaal semantics of TA composition cannot model the current extension because Uppaal requires that the guards of both synchronizing transitions should be satisfied before executing the sender action. In fact, this semantics does not evaluate $P(x)$ after the update of x by the sender.

Translation of NTACDC to NTA. In order to provide model-checking support, we give the translation of NTACDC in terms of compositional Uppaal NTA. When synchronizing, the sender copies communicated data into a shared variable that will be read by the receiver during the receive action. The resulting transition of such a synchronization will be guarded by $P(x)$.

Definition 17 (NTA corresponding to NTACDC). *Given a network of TACDC $\langle \mathcal{T}_1, .., \mathcal{T}_n \rangle$ defined on the same set of channels C, clocks χ and global variables $\mathcal{V}^g$, the NTA (with compositional semantics) corresponding to $\langle \mathcal{T}_1, .., \mathcal{T}_n \rangle$ is defined by $\langle \mathcal{T'}_1, .., \mathcal{T'}_n \rangle$ on C, χ and global variables $\mathcal{V}^g \cup \{sh\}$ where each TA $\mathcal{T'}_i$ is the translation of the TACDC $\mathcal{T}_i$ given in Definition*15.

Therefore, we have defined the semantics and translation of NTACDC through that of their individual TACDC. Similarly to TA composition, we study the compositionality of the semantics of TACDC composition.

Translation Correctness. Through the translation of NTACDC to NTA, we compare both semantics: the ETTS of a NTACDC and that corresponding to the translation of the NTACDC to a NTA, and establish the following theorem:

Theorem 7 (Bisimilarity). *The ETTS semantics of a network of TACDC is bisimilar to the ETTS semantics of the corresponding NTA.*

Proof. It relies on proving the correctness of the guards and actions of transitions produced by the rule $SR_{i,j}$. Based on the compositionality of NTA we have established, we conclude with the compositionality of the semantics of NTACDC.

Corollary 1 (NTACDC Compositionality). *The ETTS of a NTACDC is bisimilar to the restriction to time and τ-transitions of the product of ETTSs associated to individual TACDC.*

5 Conclusion

In this paper, we have studied the composition of timed systems, especially Uppaal timed automata, by considering different concepts like as communication, synchronization, variables, *etc.* In fact, we have defined an extended timed transition system with static priorities as a semantic model. Thereafter, our framework has been instantiated to define a compositional semantics for Uppaal TA composition. A translation of our compositional NTA to Uppaal NTA has been proposed with correctness arguments. Moreover, we have defined an extension enabling synchronizing TA to perform instantaneous conditional data communications. In the same way, in order to reuse the Uppaal tool, a translation of TACDC into basic Uppaal TA has been also proposed with correctness arguments. In the future, we intend to fully mechanize the work presented in this paper. This work will rely on our previous mechanization of a compositional semantics for Uppaal [8]. We wish also to extend our framework to modal specifications [20].

References

1. Alur, R.: Timed Automata. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 8–22. Springer, Heidelberg (1999)
2. Alur, R., Dill, D.: Automata for Modeling Real-Time Systems. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 322–335. Springer, Heidelberg (1990)
3. Arnold, A.: Finite transition systems: semantics of communicating systems. Prentice Hall International Ltd., Hertfordshire (1994) Plaice, J. (trans.)
4. Baier, C., Katoen, J.-P.: Principles of Model Checking. Representation and Mind. The MIT Press (2008)
5. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on Uppaal. Department of computer science. Aalborg university (2004)
6. Berendsen, J., Vaandrager, F.: Compositional Abstraction in Real-Time Model Checking. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 233–249. Springer, Heidelberg (2008)
7. Berthomieu, B., Ribet, P.O., Vernadat, F.: The tool tina – construction of abstract state spaces for petri nets and time petri nets. International Journal of Production Research 42(14) (July 2004)
8. Bodeveix, J.P., Boudjadar, A., Filali, M.: An Alternative Definition for Timed Automata Composition. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 105–119. Springer, Heidelberg (2011)
9. Clarke, E.M., Long, D.E., Mcmillan, K.L.: Compositional model checking. In: LICS 1989, pp. 353–362 (1989)
10. David, A., Håkansson, J., Larsen, K.G., Pettersson, P.: Model Checking Timed Automata with Priorities Using DBM Subtraction. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 128–142. Springer, Heidelberg (2006)
11. Garavel, H., Mateescu, R., Lang, F., Serwe, W.: CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 158–163. Springer, Heidelberg (2007)
12. Henzinger, T.A., Manna, Z., Pnueli, A.: Timed transition systems (1992)
13. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
14. Jensen, H.E., Guldstr, K., Skou, A.: Scaling up Uppaal: Automatic Verification of Real-Time Systems using Compositionality and Abstraction. In: Joseph, M. (ed.) FTRTFT 2000. LNCS, vol. 1926, pp. 19–30. Springer, Heidelberg (2000)
15. Jin, S.D., Ping, H., Sheng Chao, Q., Jun, S., Wang, Y.: Timed automata patterns. IEEE Transactions on Software Engineering 52(1) (2008)
16. Larsen, K.G., Pettersson, P., Wang, Y.: Uppaal in a nutshell. Journal on Software Tools for Technology Transfer (1997)
17. Milner, R.: Communication and Concurrency. Prentice Hall Ltd. (1989)
18. Nielsen, M., Rozenberg, G., Thiagarajan, P.: Transition-systems, event structures, and unfoldings. Information and Computation 118, 191–207 (1995)
19. Nielsen, M., Rozenberg, G., Thiagarajan, P.S.: Elementary transition systems. Theoritical Computer Science, 3–33 (1992)
20. Raclet, J.B., Badouel, E., Benveniste, A., Caillaud, B., Legay, A., Passerone, R.: A modal interface theory for component-based design. Fundamenta Informaticae 108(1-2), 119–149 (2011)
21. Yovine, S.: Kronos: A verification tool for real-time systems. Journal of Software Tools for Technology Transfer, 123–133 (1997)
22. Yovine, S.: Model Checking Timed Automata. In: Rozenberg, G., Vaandrager, F.W. (eds.) EEF School 1996. LNCS, vol. 1494, pp. 114–152. Springer, Heidelberg (1998)

On the Automated Modularisation of Java Programs Using Service Locators

Syed Muhammad Ali Shah, Jens Dietrich, and Catherine McCartin

School of Engineering and Advanced Technology, Massey University,
Palmerston North, New Zealand
{m.a.shah,j.b.dietrich,c.m.mccartin}@massey.ac.nz

Abstract. Service locator is a popular design pattern that facilitates building modular and reconfigurable systems. We investigate how existing monolithic systems can be automatically refactored using this pattern into more modular architectures, and measure the benefits of doing so. We present an Eclipse plugin we have developed for this purpose.

Keywords: static analysis, architectural antipatterns, refactoring, Java.

1 Introduction

Software modularity is an important principle in software engineering for building large and complex systems. In recent years several techniques have been introduced to facilitate software modularisation. In order to take advantage of modern modularisation techniques, many software vendors are refactoring their existing monolithic products to modular architectures. For instance, the Jigsaw project has been initiated to refactor the Java Development Kit (JDK) into a modular architecture [3]. This raises the question whether modularisation can be (semi-) automated.

Our approach is based on the assumption that modularity can be measured by the presence (or absence) of architectural antipatterns such as circular dependencies between modules [23], subtype knowledge [19], abstraction without decoupling [11] and degenerated inheritance [20]. Empirical studies [9,17] have shown that real world systems are ripe with these antipatterns, indicating a lack of modularity. In [10] we have presented an algorithm to detect critical dependencies between classes that compromise modularity. It can be shown that on the model level, the removal of these dependencies can significantly improve the modularity of the respective program. The question arises how dependencies can actually be removed or reorganised in the actual program.

The service locator pattern can be used to address this problem. In this pattern, service implementation classes are decoupled from their client classes. Fowler [12] describes the service locator pattern as a registry that is used to look up implementation classes. An alternative to manage dependencies is the closely related dependency injection pattern [12]. There is a subtle difference between service locators and the dependency injection. In the former, the client class

T. Gschwind et al. (Eds.): SC 2012, LNCS 7306, pp. 132–147, 2012.

pulls a service implementation from a registry, while in the latter the service implementation is pushed ("injected") into the client class. In this paper, we use the service locator pattern as it is less invasive - the use of dependency injection often requires the generation of additional methods (setters) and constructors in the client class.

Consider listing 1.1 as an example of the service locator pattern. In this example, the type B object is instantiated through a global factory class ServiceLocator. The method *getBImpl()* returns an implementation of the type B for the instantiation purpose. In *ServiceLocator* a setter method can be used to assign any implementation of type B. Other ways to provide the implementation type include Java Reflection and Service Registry. In particular, the use of reflection can be used to avoid *design time* dependencies from the implementation classes - the class names can be configured in meta data or configuration files, and the service locator can load those classes using reflection. This approach is widely used in frameworks such as Spring and OSGi.

```
public class A {
    private B bObject = null;
    public void m() {
            bObject = ServiceLocator.getBImpl();
            ...
    } ...
}
```

Listing 1.1. Java source code with service locator pattern

When using the service locator pattern to decouple classes, abstract types are used to declare program elements. However, we find that in practice abstract types are rarely used to declare variables and fields in a class. For example, Steimann et al. [21] presented a study that in several large Java projects only 1 out of 4 variables was declared through its interface. Therefore, we do not only have to replace constructor invocations of concrete classes by the invocation of service locator methods, but that we also have to replace declaration types by more abstract types. We have developed an Eclipse plugin for this purpose.

The rest of the paper is organised as follows: section 2 provides an overview of the related work in type generalisation and the service locator pattern. Section 3 presents a methodology to tackle the problem by explaining several refactoring processes. In the subsequent sections, we discuss experiments and results. Finally, we summarise the work and consider future directions.

2 Related Work

Martin has defined the Dependency Inversion Principle (DIP) [14], which states that "high level modules should not depend upon low level modules. Both should depend upon abstractions". This means a class should depend on an abstract type rather than on a concrete class. However, these abstract types still have to be instantiated using concrete classes. There are several ways to instantiate

a concrete class and pass it to the client class exhibiting DIP. For example, Fowler [12] has proposed to use service locators and the dependency injection for this purpose.

In the literature, several tools and techniques exist that aid developers to use more abstract types. For example, Mayer et al. [16] has developed a refactoring tool that detects code smells and executes refactorings on the source code level. In this tool the supertype hierarchy is displayed to the user in the form of a lattice. The user can choose a relevant refactoring among multiple refactoring suggestions. This process involves human interaction and cannot be fully automated.

Streckenbach and Snelting [24] have developed the KABA refactoring tool, which proposes split classes and move members refactorings. A drawback of this tool is that it cannot modify a program's source code. However, it can be used to modify the bytecode of the program. Bach et al. [6] have developed an Eclipse plugin, which finds better fitting types in programs. This plugin looks for all variable declarations, field declarations, method parameter types and method return types to compute valid supertypes. Once valid supertypes are computed, this plugin generates warnings in the Problem View of Eclipse. Quick fixes are associated with each variable declaration to automatically re-declare a variable with an abstract type. The selection of an abstract type for re-declaring a variable must be done manually. We found that this plugin is not scalable for large programs.

There are several approaches based on a metrics suite to determine the refactoring opportunities for types generalisation. Mayer [15] has analysed the use of interfaces in large object-oriented programs. This study reveals that interfaces are not very popular among programmers. The author defined a metric suite to identify source code places where the use of interfaces should be applied. For this purpose, the author developed an Eclipse plugin, which assists programmers to make use of interfaces for variable and field declarations rather than using concrete classes. In our study, we try to generalise to any compatible supertype and we don't restrict ourself to interfaces.

Gobner et al. [13] has investigated the use of interfaces in Java Development Kit. They advocate that there is a big opportunity for replacing existing types with their supertypes. The authors have validated the results with the help of several metrics that are related to the use of interfaces within the project. This study doesn't consider abstract classes but only focuses on interface utilisation. In a similar way, Steimann et al. [22] have presented a study showing that in several large Java projects 1 out of 4 variables was declared through its interface. The authors have defined a metric suite related to interface utilisation in object orient programs and proposed refactorings for a better utilisation of interfaces in programs. The metrics definition is very vague and no implementation exists to validate the correctness of the approach.

Opdyke and Johnson [18] have proposed refactorings for creating abstract super classes from concrete classes. The authors identified refactoring steps and provided techniques on how to automate these steps. Tip et al. [25] have

proposed a constraint satisfaction mechanism to verify preconditions for type generalisation refactorings such as *extract interface*, *use super type where possible* and *generalize declared types*. These refactorings have been implemented in the standard distribution of Eclipse. Steimann et al. [21] have followed a similar approach by developing a tool that can automatically infer types from concrete classes. By selecting a variable the infer type refactoring allows programmers to generate a new minimal interface that can be used to declare that variable. A disadvantage of their approach is that the excessive use of this approach may result in many similar interfaces.

3 Methodology

3.1 Antipatterns Definition

In this paper, we have focused on a set of architectural antipatterns that compromise modularity. These antipatterns compromise the separability of name spaces (packages) and separability of abstract and concrete types, therefore undermine software modularity. A more detailed discussion can be found in [9,10]. We have used visual definitions to introduce these antipatterns. Boxes inside packages represent classes and arrows represent paths (sequence of edges). Paths can traverse more than one packages. Cardinality constraints (1:M) represent the minimum and the maximum path length, where "M" is unbound.

We have used a stronger definition of circular dependency (SCD) where there is a single path that connects package1 and package2 as shown in figure 1. This definition is different from the standard definition used in many places in the literature [1,4,23] where there might not be a single dependency path that creates a dependency cycle between packages. On the other hand, SCD antipattern represents a stronger coupling between packages and cannot be easily refactored by simply splitting packages.

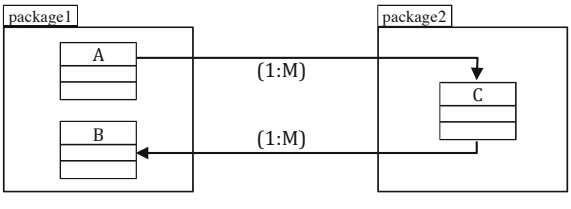


Fig. 1. Circular Dependency between Packages

In the subtype knowledge (STK) antipattern [19] as shown in figure 2 a supertype either directly or indirectly uses its subtype. This implies that we cannot use super and subtypes in isolation. In the presence of STK, separating supertypes and subtypes in different name spaces results in a circular dependency between name spaces.

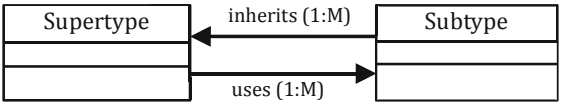


Fig. 2. Subtype Knowledge

In the abstraction without decoupling (AWD) antipattern (figure 3), a client depends on a service (supertype) and the implementation of the service (subtype) at the same time. This antipattern has several problems, such as, the client code needs to be changed whenever the service implementation changes. This situation could be avoided, had the client only depended on the service and not on the implementation. The service locator pattern can be used to avoid this antipattern.

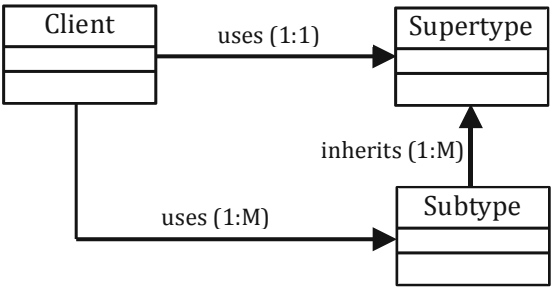


Fig. 3. Abstraction Without Decoupling

Degenerated inheritance (DEGINH) antipattern [20] as shown in figure 4 appears in a program when there are multiple inheritance paths from a subtype to its supertype. In Java programs, this can be introduced by the use of multiple inheritance through interfaces. This antipattern creates redundancy in the program structure and makes it more difficult to separate subtypes from their supertypes.

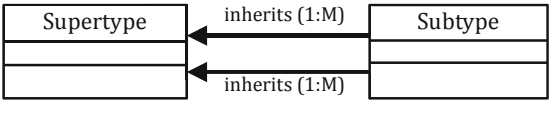


Fig. 4. Degenerated Inheritance

3.2 The Architectural Model

The architectural model we have used is the dependency graph of classes and their relationships [10]. The dependency graph is extracted from the bytecode of a program. In the dependency graph classes are represented as nodes, while edges represent relationships between classes. Dependency graphs provide an abstract representation of a program, which is useful in detecting architectural antipatterns. Several antipattern detection and refactoring tools are based on dependency graphs [2,4,7,8]. The process used to extract the dependency graph from programs is described in our previous work [9].

The tool we have used to detect architectural antipatterns instances is Guery [8]. In Guery, we define antipatterns as graph queries to be run on dependency graphs extracted from the bytecode of programs. This tool has an easy-to-use query language and a scalable implementation of the query engine that can detect antipattern instances in large programs. It provides scripting support to analyse a large set of programs [10].

3.3 Dependency Classification

Listing 1.2 shows different types of dependencies in a Java program. In this listing class A depends on other types B, C, D, E, F, G, H and System. In this context, we call class A the **source type**, and the rest of the classes **target types**.

```
public class A extends B implements C {
        private D object = new E();
        public F m(G obj) throws H {
                        System.out.println(obj.toString());
                        ...
                        return obj.getF();
    }
}
```

Listing 1.2. Java source code creating dependencies

We can broadly classify a dependency relationship between a source and a target type into the following eight categories:

1. Variable Declaration (VD): The target type is used to declare a variable or a field, for example, $A \rightarrow_{uses} D$.
2. Method Return Type (MRT): The target type is used as a return type of a method in the source type, for example, $A \rightarrow_{uses} F$.
3. Method Parameter Type (MPT): In this case the target type is used as a method parameter in the source type, for example, $A \rightarrow_{uses} G$.
4. Method Exception Type (MET): The target type is used as an exception type using *throws* keyword, for example, $A \rightarrow_{uses} H$.
5. Constructor Invocation (CI): A target type constructor is invoked through *new* keyword, for example, $A \rightarrow_{uses} E$.

6. Static Member Invocation (SMI): When a static member of a class (field or method) is invoked on the target type, for example, $A \rightarrow_{uses} System$.
7. Superclass (SC): The target type is used as a supertype through the *extends* keyword, for example, $A \rightarrow_{extends} B$.
8. Interface (IN): The target type is used as an interface through *implements* keyword, for example, $A \rightarrow_{implements} C$.

In the dependency graph, an edge can represent three types of relationships: *uses*, *extends* and *implements*. Categories from 1-6 are represented as *uses* edge. SC relationship is represented by an *extends* edge, while *implements* edge represents IN relationships. In this paper, we have tried to break dependencies from category 1-6. We have used a *replace by supertype* refactoring for first four categories, while the *introduce service locator* refactoring is used for CI and SMI categories.

3.4 The Refactoring Process - CARE Plugin

In order to execute the whole refactoring process, we have developed an Eclipse plug-in named CARE (Code and Architectural Refactoring Environment)[1]. The purpose of this plugin is to identify and execute refactorings on the source code level with a push of a button. The plugin has the ability to analyse a single program or batch-script multiple programs in an Eclipse workspace environment. Since we have built this plugin on top of existing stable refactoring tools and techniques, the refactorings we apply are proved to be safe [5,6,25].

The detailed refactoring process of the CARE plugin is shown in figure 5. CARE initially parses Abstract Syntax Tree (AST) from the source code. A refactoring is only executed if it passes pre and postconditions.

Following are main steps of the refactoring algorithm:

1. Building a dependency graph from the bytecode of a program.
2. Using the Guery engine, compute a set of SCD, STK, AWD and DEGINH instances.
3. Computing a list of high-scored edges (class dependencies) by their ranking based on their participation in all types of antipattern instances.
4. Parsing the program's source code into ASTs.
5. Checking preconditions whether a high-scored edge can be refactored.
6. If the preconditions are satisfied, apply the refactoring on the program's ASTs. Otherwise try the next high-scored edge.
7. Evaluate the postconditions to check whether the applied refactoring has introduced any errors.
8. If postconditions are satisfied, update the program source code. Otherwise, rollback the ASTs to their previous states.
9. Repeat the process until all antipatterns instances are removed or a certain number of iterations are performed (MAX is 50 in our case).

[1] http://code.google.com/p/care/

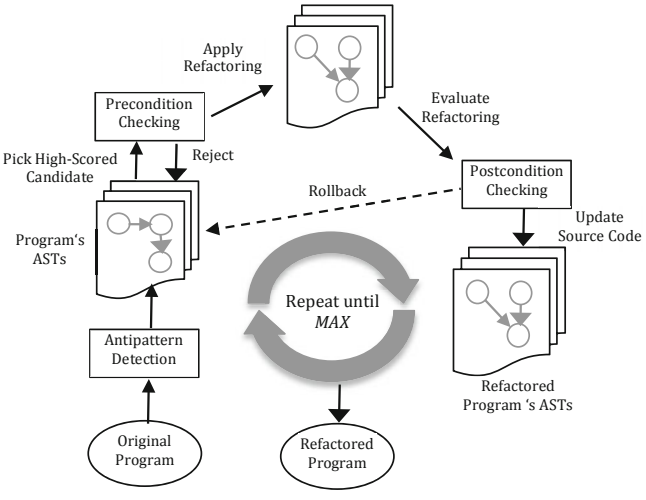


Fig. 5. Automated Refactoring Process

Scoring Function. Refactoring all antipattern instances identified in a program would completely remove the particular problem. However, refactoring all antipattern instances in large programs could be a resource demanding task. To overcome this problem we have used a scoring mechanism [10]. Scoring associates a number with each edge indicating in how many antipattern instances this edge occurs.

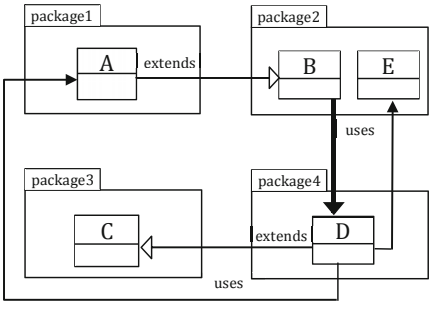


Fig. 6. Example program's dependency graph

Figure 6 shows an example of the dependency graph of a program. There are five classes in the program namely A, B, C, D, E in four packages named as package1, package2, package3 and package4. This program has two types of antipattern instances i.e. SCD and STK. The following paths represent antipattern instances:

- An SCD antipattern instance is represented by the path $A \rightarrow_{extends} B \rightarrow_{uses} D \rightarrow_{uses} A$.
- An SCD instance caused by $B \rightarrow_{uses} D \rightarrow_{uses} E$.
- An STK antipattern instance is represented by the path $B \rightarrow_{uses} D \rightarrow_{uses} A \rightarrow_{extends} B$.

In this example, the edge $B \rightarrow_{uses} D$ gets the score of three. The rest either get one or two. All of these antipattern instances can be removed either by redirecting the edge $B \rightarrow_{uses} D$ to $B \rightarrow_{uses} C$ or by removing the edge $B \rightarrow_{uses} D$.

In our algorithm we have compute those edges with the highest score. If multiple edges get the same score, we sort them by the fully qualified name of the start and the end vertex to make the process repeatable.

Parsing Source Code. After the identification of critical edges (class dependencies) that compromise modularity, we need to analyse the source code to verify whether removing or reorganising these dependencies is possible or not. For this purpose, our plugin extracts the Abstract Syntax Tree (AST) of the program. We have used the JDeodorant [5] API based on Eclipse Java Development Tool (JDT) to parse a program's source code. The plugin only parses the source code once and later if any changes are made to the source code, the modified class files are parsed through *ElementChangeListener* API of Eclipse JDT.

Refactoring Preconditions. Once we have the list of high-scored edges, we iterate over the edges until we find an edge which satisfies the preconditions. Given a source and a target type from the high-scored edge, we check the source class to make sure that all references to the target type can be replaced with references to a supertype. In case the target type is used in a CI or SMI dependency, we introduce the service locator pattern.

In CARE we have two types of refactorings: *Replace by Supertype* and *Introduce Service Locator*. These two refactorings are based on Eclipse Refactoring Toolkit (LTK). The *Replace by Supertype* refactoring is built on top of the standard Eclipse refactoring *Generalize Declared Type*. In the *Generalize Declared Type* refactoring we have to manually select a declaration type and see whether this can be replaced with one of its supertypes. On the other hand, *Replace by Supertype* automatically replaces all target type declaration elements within a class with a specific compatible supertype.

The standard Eclipse refactoring *Generalize Declared Type* computes a list of all possible supertypes with which a declaration type can be replaced. *Generalize Declared Type* is only supported on declarations of fields and variables (VD), method parameters (MPT) and method return types (MRT), but not on method exception types (MET).

The following steps describe the process through which a specific supertype for a target type T is selected:

1. For each VD, MPT and MRT type dependency to T, compute a set of possible supertypes (*PossibleTypes*) using the standard *Generalize Declared Type* plugin.
2. Extract the original supertype hierarchy of the target type *(AllSuperTypes)*. We extract this hierarchy by using Eclipse JDT Core API *ITypeHierarchy*. This API returns an array of all supertypes of the current type in bottom-up order. This means *java.lang.Object* would be the last one in the list.
3. Iterate over *AllSuperTypes* from the first *type* to the last. If the *type* exists in all *PossibleTypes* sets, choose this type as a supertype to replace the old type in all identified places of the source class and stop the iteration. If no such type exists, abandon this refactoring and try the next edge.

There are several preconditions that must be satisfied before either of the refactoring is applied on the source code level.

1. The type of a variable or a field cannot be generalized if the class members invoked on that variable or field are not part of any supertype's interface. For example, consider the following method signature in a class:
   $MyClass : int\ findMaxNumber(java.util.Vector\ v)$.
   This method signature can be refactored as follows:
   $int\ findMaxNumber(java.util.Collection\ v)$.
   The above refactoring would not work if *MyClass* invoked *v.get(i)* in the method body of *findMaxNum*. In this case, replacing *java.util.Vector* with its supertype *java.util.Collection* is not possible because the method *get(int i)* is not a part of *java.util.Collection* interface. In a similar way, we cannot generalise a type when a reference to the the target type is leaked. For instance, if we have the method *MyClass: int findMaxNumber(java.util.Vector v)*, and we have a method invocation *new OtherClass().check(v)* in the method body, then we do not know which part of the interface of *Vector* is used in *OtherClass*. While it is possible to follow this references in principle, the existing *Generalize Declared Type* refactoring does not support this.
2. The target type must not be a supertype of the source type. As we are not refactoring the class hierarchy as such, a dependency between source and target would remain in the program anyway.

There is another precondition, which is bound to the type of program we are refactoring. This precondition can be enabled or disabled depending on the program nature. If we are trying to refactor a program which is a library or a plugin extension, we should enable the following precondition. Stand alone applications that are not used programmatically by other programs do not require this precondition.

3. If a target type is used as the declaration type of a public field or used as a return type of a public method, we can't generalise that target type. Doing so may break the external client code that was dependent on that particular public field or method.

Applying the Refactoring. If the preconditions are satisfied, we can replace the occurrence of the target type with one of its supertypes and introduce the service locator pattern, if needed. In order to execute the refactorings on the source code level, Eclipse JDT Core provides an ASTRewrite API to record change modifications on a program's AST without affecting the original code. We extend the standard Eclipse *Refactoring* class to write *Abstraction Refactoring*. This refactoring encompasses both refactorings i.e. *Replace by Supertype* and *Introduce Service Locator*. Depending upon the types of dependencies the relevant refactoring is applied. Once we apply the refactoring on the source code, we get the UndoChange object. This object can later be used to rollback the refactoring if postconditions are not satisfied.

Whenever the *introduce service locator* refactoring is performed for the first time, it creates a new global factory class *ServiceLocator* in a new package named *registry*. In the case of CI type of dependency, a new method stub is automatically created in the *ServiceLocator* class and the source class is modified with a call to the newly created method. For example,
B bObject = new BImpl( ); would be replaced by
B bObject = registry.ServiceLocator.getBImpl( );
A method stub is created in the *ServiceLocator* class with the following signature: *public static B getBImpl( )*. This method would return the implementation class of type *B*. In a similar way, method stubs are created for static field and method invocations.

Refactoring Postconditions. Postconditions are evaluated in order to ensure that a program's behaviour is preserved. By default, the only postcondition checked is whether the refactored code can be compiled. Since we are using strict preconditions that can guarantee the safety of the refactorings [25], this check is de facto redundant. However, our tool can be extended to use additional project-dependent postconditions such as executing unit tests. Again, as the refactorings do not change runtime types, one could argue that those checks are redundant. However, if reflection (*instanceof*, use of *java.lang.reflect.Method*) is used in the program, the refactorings can actually change the semantics of the program and those postconditions might fail. If postconditions fail, we rollback to the previous program's state by executing the *UndoChange* object created previously and continue with the next high-scored edge in the list.

Metric Computation. A refactoring should improve a program's structure. In order to gauge the impact of refactoring on the source code level, we have defined an antipattern count metric. An improvement in this metrics would indicate the improvement before and after performing the refactoring. This metric is simply a count of the number of SCD, STK, AWD and DEGINH instances present in the program. In computing these antipattern instances, we have ignored references to and from *ServiceLocator* class, which is used to pull required dependencies into client classes. This registry class can instantiate those classes by means of reflection at runtime from names defined in configuration files or a program's meta data, therefore avoiding design time dependencies that compromise the

maintainability of the system. This approach is widely used in dependency injection frameworks and dynamic component models.

4 Results

In order to validate our approach, we have applied it to three open source Java programs Drawsf-1.2.9, JHotDraw-7.5.1 and Findbugs-1.3.9. Drawswf is a drawing application written in Java, which can be used to save drawings as flash files. JHotDraw is a framework for creating structured and technical drawings. It is heavily based on design patterns. Findbugs is a static analysis tool, which can be used to detect several code smells. These programs are relatively active, mature and suitable for academic studies. The system used to run experiments was Macbook Pro i7 2.0 GHz with 4GB RAM and Java Runtime Environment 6.

4.1 Impact of Refactorings

Figure 7 shows the decline in the total number of antipattern instances in three selected programs. The first few refactorings remove a big number of antipattern instances. This is possible if we are able to perform a refactoring based on a high-scored edge. As mentioned earlier, every refactoring is associated with several preconditions that may fail. In this case, we proceed to the lower-scored edges. Refactoring a lower-scored edge (dependency) removes fewer antipattern instances than refactoring a high-scored edge.

Table 1 shows some statistics collected from the case studies. The antipattern count represents combined instances of SCD, STK, AWD and DEGINH. The antipattern instances count is high because we have computed all variants of a particular antipattern in the dependency graph of a program. We have found

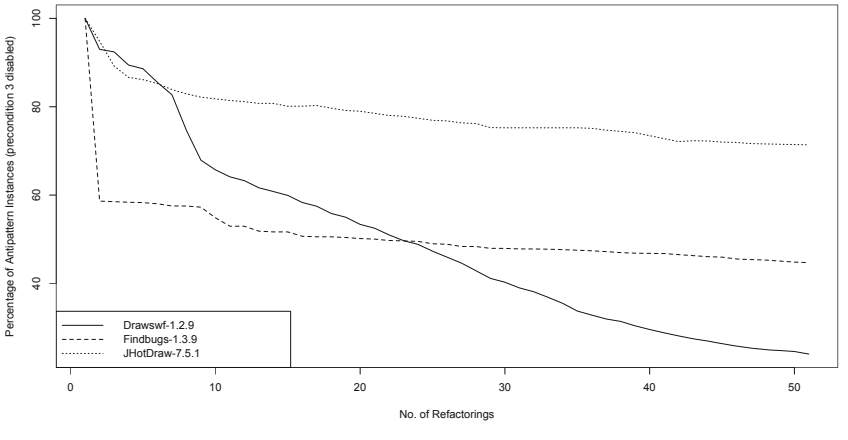


Fig. 7. Impact of Refactoring on Antipattern Count in the Three Case Studies

Table 1. Results of case studies with precondition 3 enabled

Program	Classes	Refactorings	Antipattern Before	Antipattern After	Types Generalised	SL Used
Drawswf-1.2.9	322	50	3736	661	31	110
JHotDraw-7.5.1	1193	50	7403	4947	4	116
Findbugs-1.3.9	1735	50	277091	125235	3	83

several cases where types were generalised. However, the majority of the cases belonged to the service locator pattern, as shown in the table.

Table 2. Results of case studies with precondition 3 disabled

Program	Refactorings	Antipattern Before	Antipattern After	Types Generalised	SL Used
Drawswf-1.2.9	50	3736	702	31	111
JHotDraw-7.5.1	50	7403	4946	4	115
Findbugs-1.3.9	50	277091	122585	4	81

Table 2 shows the results of the same programs but with an additional precondition, i.e., we do not refactor public fields and return types of public methods. There is a slight difference between the results of two experiments in terms of the number of types generalised and the usage of service locator pattern. This indicates that not many public methods' return types and public fields were modified in either of the experiment. This additional precondition, however, affected the total number of antipattern instances removed after 50 refactorings.

4.2 Discussion

In this paper, we have tried to either remove or reorganise a class dependency that causes modularity problems. In general, it is not possible to remove all types of dependencies. However, there are certain cases where it is possible to safely remove undesired dependencies. For example, the first critical dependency identified in Findbugs-1.3.9 is the reference from *edu.umd.cs.findbugs.ShowHelp* to *edu.umd.cs.findbugs.gui.FindBugsFrame*. This dependency is caused because *ShowHelp* is invoking a static method *showSynopsis( )* on the target type *FindBugsFrame*, as shown in the listing 1.4. This particular dependency is also a reference from a core package *edu.umd.cs. findbugs* to a presentation package *edu.umd.cs.findbugs.gui* and therefore violates the design principle that logic should not depend upon presentation. This particular dependency was involved in 113579 SCD instances, 185 STK instances and 1811 AWD instances. Therefore, removing this dependency reduced the total number of instances from 277091 (100%) to 161517 (58%).

```
public class ShowHelp {
 public static void main(String[] args) { ...
  FindBugsFrame.showSynopsis();
     ...
  } ...
}
```

Listing 1.3. Source code ShowHelp

The refactored version of the old code causing the dependency is as follows: *registry.ServiceLocator.initialiseSMIFindBugsFrame_showSynopsis()*. A method stub is automatically created in the *ServiceLocator* class and the old reference in the source class is replaced by a call to this method.

Another example of a critical dependency identified in JHotDraw is a reference from *org.jhotdraw.draw.TextFigure* to *org.jhotdraw.draw.tool.TextEditingTool*. This dependency was involved in 34 SCD instances and 7 AWD instances. The listing 1.4 shows the variable declaration reference in the *TextFigure* that causes this dependency. This particular dependency is a clear violation of the Dependency Inversion Principle because *TextFigure* is using an abstract type *Tool* (an interface) as well as a concrete type *TextEditingTool* (an implementation class of *Tool*). This dependency is safely removed by abstracting the old variable declaration type to *Tool* and by replacing the constructor invocation call with a call to *ServiceLocator* to obtain the implementation class of the interface *Tool*.

```
public TextFigure ... { ...
  public Tool getTool(Point2D.Double p) {
  if (isEditable() && contains(p)) {
   TextEditingTool t = new TextEditingTool( );
   return t; }
  return null;
  } ...
}
```

Listing 1.4. Source code TextFigure

All the code level changes performed by the plugin are recorded to be reviewed by a developer of the program. We suggest to use this tool in conjunction with a code repository such as a Subversion client, in case the developer wants to rollback a particular refactoring. Standard metrics can be used to check for other changes introduced by the refactoring.

5 Conclusion

In this paper, we have presented an Eclipse plugin that can identify and execute potentially high-impact refactorings on the source code. We have used a set of four architectural antipatterns to identify refactorings that compromise software modularity. The automated refactoring process, presented in this paper, removes a large number of these antipattern instances from programs. The approach was

applied on three open source programs. The results show a substantial improvement in terms of decrease in the total number of antipattern instances. However, it is possible to improve these results further. For instance, with the reference leaking issue our preconditions are very strict and restrictive. By strengthening the reasoning over the parts of the interface of a class that is actually being used in the program, we can expect to improve the presented results significantly, i.e., more refactorings could be safely executed.

We have demonstrated that process of refactoring can be fully automated. With the help of pre and postconditions it is possible to achieve the safe execution of refactorings. We have used the type-safety mechanism through compilation to ensure that the automated refactoring process did not introduce any errors in the source code of programs. However, in future we intend to use other mechanisms such as unit testing to validate the correctness of refactoring. The idea is to execute a set of testcases before and after performing refactorings. We also intend to extend the *Introduce Service Locator* refactoring by automatically adding an instantiation mechanism (Service Registry, Java Reflection) in the method stubs created in the registry.

There are several refactoring techniques to deal with undesired dependencies in programs, such as, moving classes, splitting packages, object inlining etc. It would be interesting to explore a combined search-based architectural refactoring to deal with modularisation problems discussed in this paper.

References

[1] CrocoPat, http://code.google.com/p/crocopat/
[2] JDepend Dependency Analyser, http://clarkware.com/software/JDepend.html
[3] Project jigsaw, http://openjdk.java.net/projects/jigsaw/
[4] Restructure101. Headway software technologies, http://www.headwaysoftware.com/products/?code=Restructure101
[5] Bad smell identification for software refactoring (2007), http://www.jdeodorant.org
[6] Bach, M., Forster, F., Steimann, F.: Declared Type Generalization Checker: An Eclipse Plug-In for Systematic Programming with More General Types. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 117–120. Springer, Heidelberg (2007)
[7] Bischofberger, W., Kühl, J., Löffler, S.: Sotograph - A Pragmatic Approach to Source Code Architecture Conformance Checking. In: Oquendo, F., Warboys, B.C., Morrison, R. (eds.) EWSA 2004. LNCS, vol. 3047, pp. 1–9. Springer, Heidelberg (2004)
[8] Dietrich, J., McCartin, C.: Scalable motif detection and aggregation. In: Australasian Database Conference, ADC 2012. CRPIT, vol. 124, pp. 31–40. ACS, Melbourne (2012), http://crpit.com/confpapers/CRPITV124Dietrich.pdf
[9] Dietrich, J., McCartin, C., Tempero, E., Shah, S.M.A.: Barriers to Modularity - An Empirical Study to Assess the Potential for Modularisation of Java Programs. In: Heineman, G.T., Kofron, J., Plasil, F. (eds.) QoSA 2010. LNCS, vol. 6093, pp. 135–150. Springer, Heidelberg (2010)

[10] Dietrich, J., McCartin, C., Tempero, E., Shah, S.M.A.: On the existence of high-impact refactoring opportunities in programs. In: Australasian Computer Science Conference, ACSC, Australia (2012), http://crpit.com/confpapers/CRPITV122Dietrich.pdf
[11] Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
[12] Fowler, M.: Inversion of control containers and the dependency injection pattern (2004), http://martinfowler.com/articles/injection.html#InversionOfControl
[13] GoBner, J., Mayer, P., Steimann, F.: Interface utilization in the java development kit. In: Proceedings of the 2004 ACM Symposium on Applied Computing, SAC 2004, pp. 1310–1315. ACM, New York (2004)
[14] Martin, R.: Object oriented design quality metrics: An analysis of dependencies. Report on object analysis and design (1994), http://www.objectmentor.com/resources/articles/oodmetrc.pdf
[15] Mayer, P.: Analyzing the use of interfaces in large OO projects. In: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003, pp. 382–383. ACM, New York (2003)
[16] Mayer, P., Meissner, A., Steimann, F.: A visual interface for type-related refactorings. In: WRT 2007, pp. 5–6 (2007)
[17] Melton, H., Tempero, E.: Identifying refactoring opportunities by identifying dependency cycles. In: Proceedings of the 29th Australasian Computer Science Conference, ACSC 2006, vol. 48, pp. 35–41. Australian Computer Society, Inc., Darlinghurst (2006)
[18] Opdyke, W.F., Johnson, R.E.: Creating abstract superclasses by refactoring. In: Proceedings of the 1993 ACM Conference on Computer Science, CSC 1993, pp. 66–73. ACM, New York (1993)
[19] Riel, A.J.: Object-Oriented Design Heuristics. Addison-Wesley, Boston (1996)
[20] Sakkinen, M.: Disciplined Inheritance. In: ECOOP 1989: Proceedings of the 1989 European Conference on Object-Oriented Programming, pp. 39–56 (1989)
[21] Steimann, F., Mayer, P., MeiBner, A.: Decoupling classes with inferred interfaces. In: Proceedings of the 2006 ACM Symposium on Applied Computing, SAC 2006, pp. 1404–1408. ACM, New York (2006)
[22] Steimann, F., Siberski, W., Kuhne, T.: Towards the systematic use of interfaces in java programming. In: Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java, PPPJ 2003, pp. 13–17. Computer Science Press, Inc., New York (2003)
[23] Stevens, W., Myers, G., Constantine, L.: Structured design, pp. 205–232. Yourdon Press, Upper Saddle River (1979)
[24] Streckenbach, M., Snelting, G.: Refactoring class hierarchies with kaba. In: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, pp. 315–330. ACM, New York (2004)
[25] Tip, F., Kiezun, A., Bäumer, D.: Refactoring for generalization using type constraints. In: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, OOPSLA 2003, pp. 13–26. ACM, New York (2003)

Touching Factor: Software Development on Tablets

Marc Hesenius, Carlos Dario Orozco Medina, and Dominikus Herzberg

Department of Software Engineering
Faculty of Informatics, Heilbronn University
Max-Planck-Str. 39, 74081 Heilbronn, Germany
{marc.hesenius,dominikus.herzberg}@hs-heilbronn.de,
corozcom@stud.hs-heilbronn.de
http://www.hs-heilbronn.de/

Abstract. Mobile devices have been making their way into our everyday life for quite some time, and especially the market for tablets is increasing. They are used at home for entertainment purposes as well as in professional environments, helping to ensure productivity. A large and constantly growing amount of apps for basically every task is available. With one exemption: programmers are still bound to the classic PC setup and hardly use a tablet for software development. The reason for this is simple: their tool chain does not fit the small screens. The space constraints on tablets demand a paradigm shift. We propose a prototype sketch of a development environment based on a concatenative programming language. Concatenative programming has a strong focus on composing words out of other words, supporting a minimalistic and concise approach to programming. This approach perfectly fits into the mobile world and allows developers to write programs for tablets *on* tablets.

1 Introduction

For the last years, smaller and more powerful technological devices have constantly appeared on the consumer market – a trend not likely to stop. These devices support and comfort our mobility, keeping us connected, productive and informed at any time and almost everywhere. The impact that smartphones and tablets have on the way we interact with computers is already noticeable; touching and sliding, dragging and dropping non-physical objects on a screen has become a common way of handling and interacting with applications. These new devices have their own kind of constraints and limitations, especially the restricted space due to the smaller screen. Although familiar interfaces like a static keyboard are making their way into the tablet world (connected e.g. via Bluetooth), we will focus on a scenario without peripherals, just the built-in features are considered.

T. Gschwind et al. (Eds.): SC 2012, LNCS 7306, pp. 148–161, 2012.

The market especially for tablets is growing heavily.[1] Several hundred thousands of applications are nowadays available for nearly every need; little helpers for everyday tasks as well as complex business software can be found. Oddly enough, the economically most important creative workforce is left out: no professional development environment is available and programmers hardly use a tablet to develop software. Classical PCs are a developer's main workplace, often with several monitors. It seems strange – one can create software *for* tablets but not *with* tablets.

Software development is dominated by languages like Java, C and all its derivatives.[2] All of them are grammar-heavy, imperative programming languages with many syntactical conventions, resulting in big blocks of text to create a program. In some way, they force the developer to have a large working space. Modern Integrated Development Environments (IDEs) support this need by providing an infrastructure similar to a complex command center – their screen is fragmented into lots of small windows providing different views and concerns for browsing, navigating, editing, debugging etc. Having these needs of space in mind, developing software using a mobile device does not seem to be feasible at all or efficient enough to be productive. Furthermore, mainstream languages do not provide the necessary environment for developing in an interactive way.

As a counterpoint, alternative programming paradigms exist, paradigms having a remarkable minimalism combined with high productivity. For example, concatenative or functional programming languages have a completely different approach to software development. Promoting an interactive programming style, they give developers the chance to work with the program in a more connected way and reduce the mental transition during coding. Furthermore, they do not ship with the built-in need for a large and massive IDE.

It is clear that today's imperative programming languages can hardly fit the needs of a reduced yet highly mobile environment that is offered by a tablet. We propose an alternative approach allowing a productive way of developing software on a mobile device. Addressing the basic requirements we have to fulfil in order to create a programming environment suitable to productive working, three main problems will be examined:

- Identification of a suitable programming paradigm fitting the needs of mobile devices
- Proposal of an eligible and productive developing environment
- Solving issues specific to the chosen paradigm regarding the user interface

Section 2 takes a deeper look into the concept of developing software on a mobile device and provide some reasoning why it is interesting to do so. It will also deal with the restrictions given by space constraints, which eventually leads to the selection of a suitable programming paradigm. Section 3 examines concatenative programming and points out the advantages of this programming paradigm over the languages used in today's mainstream software development. Section 4

[1] http://www.gartner.com/it/page.jsp?id=1626414, Jan 2012
[2] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, Jan 2012

presents a graphical prototype sketch according to the requirements defined in Sec. 2. Section 5 gives an overview of the current research situation. Section 6 presents our conclusion and gives a perspective on future work.

2 Productive Programming on a Tablet

The use of mobile devices has experienced an enormous increase over the last years. The tablet market has grown way beyond any analyst's prediction, increasing almost about 300% from 2010 to 2011 [1]. This growth is – as a logical consequence – accompanied by a growing number of applications (so-called apps), which are getting more and more robust as developers learn to cope with the special requirements of mobile devices. Activities like image and video editing or document processing are common tasks with a tablet – instead of mere passive media consumption, they are increasingly used for active content creation [10].

What makes working on a tablet so interesting? First of all, the mobility aspect is obvious. Tablets are handy devices, made for carrying them around. They are lightweight, optimized for running a long time on battery and they allow to be online almost everywhere. Today, with the capabilities of internet-based applications still increasing, the availability of information and the possibilities offered by social networks are important to people. Mobile devices are tailored to fit these new requirements. Tablets and smartphones are also having a major impact on the way we work with computers nowadays, changing how we handle applications and data. Equipped with a touch screen, tablets provide a more interactive and intuitive approach to working with a device than classic PCs do. Moving an object around with a finger is a complete different experience than using a mouse – more natural and more tactile. Mobility and interactivity are the two core concepts making tablets interesting for home as well as professional scenarios.

Some of the capabilities mobile devices ship with could also enhance the development process, changing the way of how software is actually created – especially in modern software development dominated by agile methods. Besides the possibility to program practically everywhere, mobile devices are also capable of connecting in no time to other devices, hence supporting just-in-time collaboration. Newly created functions could be distributed on the fly between different people and sharing a workspace with colleagues would be easy and straightforward.

The usage of mobile devices in the industry is increasing,[3] and professionals from different industrial branches enjoy more freedom and mobility. However, software developers are left out. Programmers seem to be tied to their PCs as their main working environment are tremendous IDEs like Eclipse or MS Visual Studio, which provide a window for many different aspects such as packages, editor, debug and test window, console for error messages, version management, etc. It is common for developers to spread this pile of information across

[3] http://www.zew.de/de/presse/1837, Jan 2012

several monitors. As shown by [4], the size of the screen makes a big difference in productivity.

A tablet has a significantly smaller screen – hence the need for a more minimalistic and concise environment. With this in mind, having a development environment on a tablet could be seen as a hard to achieve task. But the main reason for this is not the idea itself, it is the set of tools used. Imperative programming languages require a significant amount of syntactic load to create statements and structure code, therefore they exceed the restrictions of the workspace on a tablet. The strict separation of design and run time typical for current mainstream programming languages makes interactivity difficult. This obstacle manifests itself as the well-known build-deploy-run cycle – programmers usually write code, click a button to compile and execute the code, and then examine the outcome. As a consequence, software developers are forced to work with a mental model of the program's execution when actually writing it and need additional tools like debuggers to establish a connection between both of them.

Alternative paradigms help overcome these obstacles. Concatenative programming languages like Forth have interesting traits making them attractive for programming on tablets, as their minimalistic and concise approach fits into a reduced space environment. A suitable programming language must have compact yet understandable code; a concise syntax does not only save space, it also supports a developer's productivity by simplifying writing and reading a program. Code written in Forth is not only significantly smaller than corresponding programs in other languages, the visible stack also supports understandability for the programmer [6]. Furthermore, concatenative programming languages support interactive programming by allowing developers to directly communicate with the run time environment.

3 Advantages of Concatenative Programming

Concatenative languages support composition of functions (or *words* as they are called), leading to a more concise way of coding. Any sequence of words can practically be transformed into and replaced by a new word, without the need of having to take common problems like nested scopes into account. This is for a simple reason: in contrary to mainstream languages, which apply a function to a set of parameters (thus we call these languages *applicative*), concatenative languages pass a data structure with all needed information from word to word. The notion of a stack is a helpful metaphor, hence the close relationship to stack-based languages. The resulting function call can be described in simple terms: the function receives the stack as an input, pops elements of it, does some calculation, pushes the results (if any) back on the stack and finally returns the modified stack [8]. Words can return multiple values by just pushing several values back on the stack. Furthermore, this kind of processing has a couple of benefits for our main purpose of bringing a full-fledged programming environment to tablets. Outstanding is the lack of variables. Source code can be written in a point-free manner, which reduces the amount of written text.

Furthermore, without variables there is no need of taking nested scopes into account, making programming easier by removing intellectual barriers and supporting refactoring. Generally, the usage of the stack in concatenative languages is one of the reasons for the conciseness and simplicity of the code. The resulting postfix notation might be uncommon to programmers coming from the mainstream languages. Postfix notation supports minimalistic syntax and helps to go without e.g. parenthesis as they are common in Lisp dialects.

The composition abilities of concatenative languages also support an interactive approach to software development. Most current mainstream languages work on two levels: the design level for writing code on the one hand and the run time level for executing code on the other hand. Developers have to make a mental transition from design to run time when writing the actual source code, an approach that can barely be called interactive. As seen before, interactivity is a crucial component in the way a tablet is handled, therefore a possible programming environment must also be highly interactive and the chosen language should support this style of development. For starters, the approach quite many functional programming languages use is of interest. They often ship with a special kind of terminal, the so-called *Read-Evaluate-Print-Loop* (REPL). The REPL provides a run time environment as well as the possibility to directly define new functions; developers can connect to and modify the run time straight away. However, the use of a REPL does not exhaust the possibilities of interactivity. Although it moves developers closer to the run time, the effects of code modification are only visible when the function in question is called manually. To match the interaction capabilities of a touch screen, a more sophisticated way is desirable. Developers should be notified about the consequences of their code when actually writing it.

To sum up, we discussed the core concepts of tablets and the constraints they impose on a suitable development environment as well as their benefits. We have seen that, to bring programming to mobile devices, we have to reduce the amount of text written to fit the smaller screen but we also have to ensure an interactive way of programming. Furthermore, we have seen that current mainstream programming languages do not suffice these requirements, but that alternative programming paradigms particularly suited for our purpose exist.

We propose to move away from grammar-heavy imperative programming and claim that concatenative programming is well-suited to space constraints as well as interactivity needs. On the one hand, minimalism is combined with simplicity and conciseness, and on the other hand, concatenative programming supports interactive working with source code. The proposed programming language for our prototype is Factor[4], a recent concatenative language enhanced with features from other programming paradigms [11]. For example, Factor brings ideas from object-orientation as well as functional programming into the concatenative world and ships with a strong metaprogramming model. Furthermore, Factor supports an interactive way of software development by allowing code reflection and introspection as well as reload at run time. Prog. 2 shows an

[4] http://factorcode.org/

implementation of the well-known Quicksort algorithm in Factor. Compared to its Java sibling, which is shown in Prog. 1, this example demonstrates the amount of space gained.

Program 1. Quicksort in Java[a]

```
public static <E extends Comparable<? super E>> List<E>
                                          quickSort(List<E> arr) {
  if (arr.size() <= 1)
    return arr;

  E pivot = arr.get(0);

  List<E> less = new LinkedList<E>();
  List<E> pivotList = new LinkedList<E>();
  List<E> more = new LinkedList<E>();

  // Partition
  for (E i : arr) {
    if (i.compareTo(pivot) < 0)
      less.add(i);
    else if (i.compareTo(pivot) > 0)
      more.add(i);
    else
      pivotList.add(i);
  }

  // Recursively sort sublists
  less = quickSort(less);
  more = quickSort(more);

  // Concatenate results
  less.addAll(pivotList);
  less.addAll(more);
  return less;
}
```

[a] http://rosettacode.org/wiki/Sorting_algorithms/Quicksort#Java

While most readers are likely to be familiar with the Java version, the implementation of the quicksort algorithm in Factor requires some explanations. In Prog. 2, the colon introduces a word definition, here `qsort`. In parenthesis, the stack effect is given. Word `qsort` consumes a sequence from the top of the stack and returns a sequence on the stack. The words following the stack effect description up to the semicolon define the meaning of `qsort`. Words embraced in squared brackets are called quotations, which represent word sequences for deferred execution.

Program 2. Quicksort in Factor[a]

```
: qsort ( seq -- seq )
    dup empty? [
      unclip [ [ < ] curry partition [ qsort ] bi@ ] keep
      prefix append
    ] unless ;
```

[a] http://rosettacode.org/wiki/Sorting_algorithms/Quicksort#Factor

The Factor code is a very readable and instructive description of the quicksort algorithm: `unless` the `dup`licate of the topmost stack element (which is the sequence to be sorted) is `empty?` the code within the quotation (from `unclip` to `append`) is executed. The first element of the sequence is `unclip`ped, which is the so-called pivot element. The following quotation (from `[ < ]` to `bi@`) works on the remaining sequence, from which the pivot element has been removed, but `keep`s the pivot element on top of the stack. Whilst `curry` is somewhat special and remains left without further comments, the sequence is `partition`ed into two sequences, based on the criteria, whether elements are `<` (smaller) than the pivot element or not. Word `bi@` calls `qsort` on both partitions, which is the recursive part of the quicksort algorithm. When the calls are done, the pivot element (which has been kept topmost) is `prefix`ed to the second sorted sequence; after that both sorted sequences are `append`ed, which is the result of `qsort`.

The interested reader might consult the Factor documentation[5] to get more information on the meaning of individual words and their stack effects. While beginners of concatenative languages tend to be focused on stack effects, advanced programmers aim to write almost literate code of documentary quality.

4 Graphical Interface Prototype

To provide a solution on the restricted space problems mentioned before, we will focus on sketching a possible graphical interface capable of handling the space constraints. Furthermore, we will solve some Factor related issues, e.g. how *vocabularies* (a similar concept as libraries in imperative languages) will be handled by the interface. We examine how the debug process can be implemented in this workspace and how the navigation between words can be accomplished. To achieve these goals, our prototype consists of four main components:

1. The dictionary, which represents our vocabularies
2. The scratchpad, the developer's workspace
3. The traveller, a companion on the developer's journey through his code
4. The stack viewer, an all-time information source

Figure 1 shows the default configuration of our prototype; the above numbers refer to the areas indicated. All four components are active and positioned in the basic layout.

[5] http://docs.factorcode.org

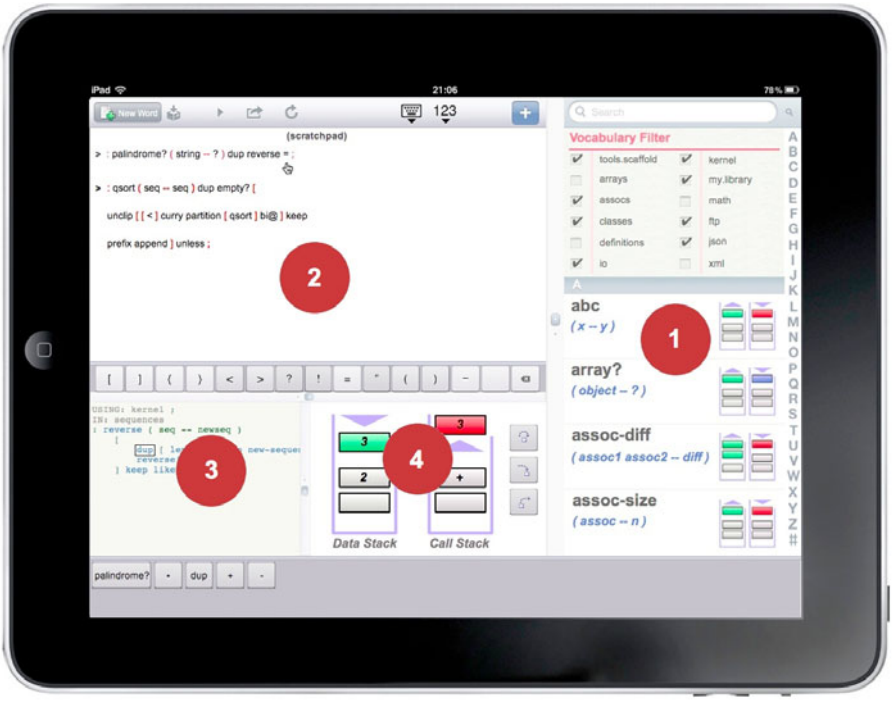


Fig. 1. Prototype default configuration

4.1 The Dictionary

Working with a huge number of words is a complicated task when the available space is limited; even more so having in mind that the number of words may increase dynamically. This is naturally the case when developers add new words or even complete vocabularies to their program. A dictionary helps the developer in *housekeeping* vocabularies and the words within. Using a dictionary is intuitive, the words are organized alphabetically, permitting to navigate between them in a simple way by just touching the letter the developer wants to see or use.

The dictionary's behavior is enhanced with some special features to allow simpler interaction. First of all, there is a section inside the dictionary called *vocabulary filter*. Every time a vocabulary is added to the dictionary, its name is added to the filter. This allows to view or hide the words of the specific vocabulary inside the dictionary using a checkbox.

Furthermore, a drag-and-drop feature allows the developer the use of a specific word on the scratchpad. A word can be dragged from the dictionary and simply be dropped onto the scratchpad wherever needed, supporting a faster development process. Alternatively, a word can only be touched and released and will automatically be inserted at the cursor's position.

In the top bar of the dictionary, as shown in Fig. 2, a search bar is located. This search functionality allows the developer to search for specific words inside the whole dictionary helping him to directly and quickly access words.

Additionally, each word in the dictionary is accompanied by a graphical and textual representation of its *stack effects*, which is a Factor term describing the number and type of items consumed and produced from/on the stack. The textual representation corresponds to Factor's built-in *stack effects notation*, while the graphical stack representation uses green for consumed and red for produced elements. If the word puts *true* or *false* after the execution on the stack, a blue block will appear on top of the stack. If the word does neither consume nor produce elements, the stack is shown as empty.

Fig. 2. The dictionary

4.2 The Scratchpad

The scratchpad is where the developer creates the software itself. It is, in simple words, a large space where all the statements are condensed to create pieces of Factor code to reuse later as words. As well as the dictionary, the scratchpad ships with some enhancements to ease the developer's life and increase productivity: first of all, a list of special symbols is shown below the scratchpad. These symbols help reduce the usage of the normal keyboard of the tablet to a minimum. The developer will be able to place these symbols immediately on the scratchpad at any time by just touching them. Furthermore, the scratchpad uses syntax coloring as shown in Fig. 3 to facilitate code visualization.

The scratchpad indicates the position where the next word or symbol will be placed by a cursor. This cursor can be moved using a finger at any time. Having this extra information inside the scratchpad helps to identify where new words are placed. However, words can be moved via drag-and-drop anywhere on the scratchpad.

Words already placed on the scratchpad can be duplicated via a multi-touch gesture. If the fingers are left on the screen, the duplicate can be dragged to another position and simply dropped there, while it will be auto-inserted at the cursor position as soon as the fingers leave the screen.

The top bar of the scratchpad, see in Fig. 3, contains some additional functionality to help the developer with his work. From left to right, the top bar offers the following features:

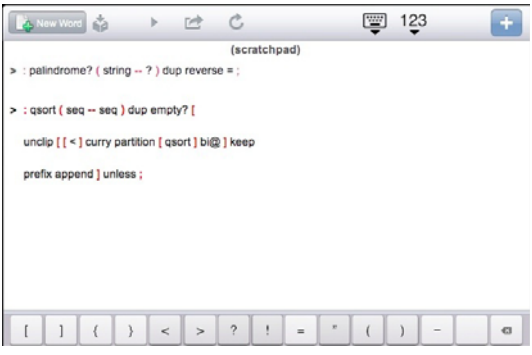


Fig. 3. The scratchpad

- **New word button:** Initializes the creation of a new word from scratch. A stub for a word in Factor is automatically created.
- **Save button:** Saves the word from the scratchpad. If the word does not have any errors, it is stored and the scratchpad is cleared.
- **Execute button:** All statements on the scratchpad will be executed. The stack viewer will show the status of the data stack after execution.
- **Debug button:** Starts the debugging process for the statements inside the scratchpad.
- **Last word button:** Shows the last saved word on the workspace again.
- **Show keyboard:** Gets the tablet keyboard on the screen hiding parts of the dictionary, the traveller (see Sec. 4.3) and the stack viewer.
- **Show numbers:** Shows the numbers keyboard.
- **Add:** Allows the user to add an already built vocabulary to the dictionary or also to create a new one.

As an addition to the scratchpad, our prototype ships with a *most common words* section, which is shown at the bottom of Fig. 1. This section allows the developer to quickly access frequently used words on a per-session base.

4.3 The Traveller

In Factor, code is composed by words which at the same time are made of other words, thus creating a highly recursive navigation issue. Our prototype solves this situation by using a fixed space only for navigation purposes. This space is called the *traveller*.

The traveller supports navigating through the definitions of words, creating a path to follow among words. Our prototype permits to click on each and every word inside the scratchpad to start the *trip*; every time more information (of a definition) of a word is needed, the developer touches the word on the scratchpad and the traveller will show its definition. On the traveller space, all those words used in the definition will be surrounded by a black box, as shown in Fig. 4 on

the left side. The black box indicates that the word could also be touched to get the definition.

After travelling inside the words, the developer can go back by just using a *swipe* gesture inside the traveller space to go one step back.

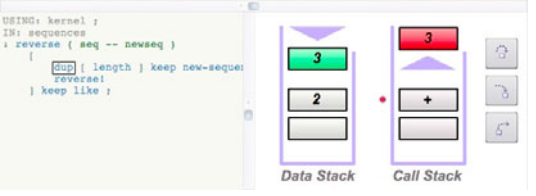


Fig. 4. The traveller and the stack viewer

4.4 The Stack Viewer

The final section of the prototype is called the *stack viewer*. In this part of the prototype two stacks are shown, the data stack and the call stack. Here the developer has the option of looking at the stack status after executing the code in the scratchpad. Factor itself uses a textual representation of the stack, but we propose a visualization to clarify stack behavior. This should especially be helpful for developers coming from an imperative language. After executing the code inside the scratchpad, the data and the call stack will be shown with their resulting status. To examine all elements on the stack, developers can simply scroll the stack content up and down.

The stack viewer can also be used to monitor effects on the program when actually coding. Whenever the developer makes (valid) changes to the code, the stack viewer is updated and reflects those changes. This is a real-time stack effect inference engine. By this means, the developer is supported by reducing the need to create a mental model of the run time – the results are immediately visible.

The stack viewer can also be used for debugging purposes. When the debug procedure is called from the scratchpad, a step-by-step activity will be launched in the stack viewer. The developer can now examine the status of both stacks stepwise, seeing what is pushed and what is popped. Adding breakpoints inside the call stack is also possible to allow the developer to see the stack behavior for specific words or a specific moment in the programm execution.

Three buttons are added inside the stack viewer during the debug process:

- **Step:** Move the debug process one step forward.
- **In:** Allows the user to go inside a word definition and stack status.
- **Out:** Allows the user to go back to the main word definition and stack status.

5 Related Work

At present, the connection between software development, a PC, a keyboard and one or more displays seems to be fix and unbreakable. Research in this area is

quite young and has merely just begun. First attempts of creating a suitable development environment on tablets are surfacing. McDirmid proposed an interesting idea built around the visual programming language YinYang in [10]. He showed how tablets can compete with classic PC setups when it comes to programming. Despite its benefits, visual programming has not gained a lot of acceptance – most programmers still work with text-based environments. Therefore, we do not follow the idea of visual programming. Instead we suggest to stick to the familiar, text-based working approach and enhance it with the advantages of a device with touch capabilities. Tillman et al. introduced TouchDevelop in [12], a less visual and more textual oriented approach. TouchDevelop focusses on different aspects than we do, aiming at students and hobbyists instead of professional developers and being created mainly for smartphones. Personalization of the phone and fun are the main objectives. TouchDevelop as well as McDirmids development environment also use programming languages specifically created for their purpose.

Codea[6] (formerly known as Codify) is a development environment created for the iPad. It is built around the Lua language and shows interesting ideas in the matter of how developers can interact with their code. Lua is an imperative language, and also the demo video implies the use of a keyboard. Our prototype does not need any peripherals.

AIDE[7] is the attempt to bring an IDE to Android to support the creation of apps directly on an Android-based device. AIDE sticks to Java, and it should be rated as a show-case that an IDE can be ported to Android systems; the usability for productive programming is by all means doubtful.

Having the lack of available research in mind, the surrounding boundaries become of interest, e.g. the usability approaches and issues on mobility presented by Jung [9] as well as Xu and Bradburn [13]. The concept of visual programming has been around for years with interesting ideas like the navigation solution from Edel [5] and the usability concept created for kids by Chen, Wang and Wang [3].

As most current programming languages will not go very well with the reduced space environment on a mobile device, we propose to think about alternative programming paradigms. We need a minimalistic yet powerful programming language capable of coping with the restrictions on a tablet. These requirements are fulfilled by the family of concatenative programming languages, which in the earliest incarnation Forth have been around quite a while [2]. In our prototype sketch, we use Factor presented by Pestov et al. in [11] because it ships with all needed advantages from concatenative programming and adds some interesting features from other language families. An overview of the concatenative paradigm is provided by Herzberg and Reichert, accompanied by the implementation of Concat as an example in [8]. A first glimpse of how the combination of concatenative languages and mobile devices can be used in learning environments to support prospective programmers in their studies is presented by Herzberg, Hesenius and Reichert in [7].

[6] http://twolivesleft.com/Codea/, Mar 2012
[7] https://play.google.com/store/apps/details?id=com.aide.ui, Mar 2012

6 Conclusions and Perspective

Creating new IDEs for mobile devices is not enough to ensure their efficiency for productive software development. The space restrictions and the usability requirements for software developers are constraints that cannot be satisfied with imperative languages, hence different ideas are needed. Mainstream languages with the static and ever recurring build-deploy-run cycle stand in contrast to the interactive possibilities provided by a touch screen. We proposed an alternative approach, using the concatenative programming paradigm, and sketching a prototype to provide a solution for some of the problems related to software development on a tablet.

Our prototype demonstrates how a concatenative programming language can be used to solve the problems with the reduced screen size on a tablet. We used Factor as an example, which besides a minimalistic approach has a number of interesting traits and features making it suitable for being the first choice programming language for our purpose. We showed ideas of how the navigation issues within vocabularies can be solved, how a debug process can be implemented and how the vocabularies and words could be organized, helping the developer to be more productive. Furthermore, Factor ships with built-in features offering a wide range of capabilities for interactive working with source code.

For future work some open questions and tasks remain. In an environment without a keyboard, code completion and code suggestions are useful and welcome helpers – this problem can easily be solved looking up names of already created words or investigating the stack effects of words to try to find similar words behavior. The tablet's built-in keyboard can be used in a more context-sensitive way, changing its layout according to the developer's current task or goal. A more sophisticated system of intelligent suggestions is of interest, which supports the developer and finds the right word when needed. In certain situations, the graphical stack representation we proposed does not suffice, therefore some words cannot be represented this way; there is a need for a more sophisticated stack visualization. The interaction abilities of a tablet are by far not exhausted by our prototype. A first step can be done by having the current stack always visible; changes to the code should modify this stack in an instant. To ensure more interactivity, a close connection to the run time is needed. Developers should see the consequences of changes in the running application in the moment they alter the code as well as be able to directly manipulate the run time by e.g. moving objects around.

Acknowledgements. We thank the Thomas Gessmann-Stiftung for partially supporting this paper.

References

1. Bibby, A.: The rapid development of tablet computing. White Paper 3.0, TCO Development (October 2011)

2. Brodie, L.: Thinking Forth. Punchy Publishing (2004)
3. Chen, X., Wang, D., Wang, H.: Design and implementation of a graphical programming tool for children. In: 2011 IEEE International Conference on Computer Science and Automation Engineering, CSAE (2011)
4. Consulting, P.: The 30-inch apple cinema hd display productivity benchmark. Tech. rep., Pfeiffer Consulting (2005)
5. Edel, M.: The tinkertoy graphical programming environment. IEEE Trans. Softw. Eng. 14, 1110–1115 (1988)
6. Ertl, M.A.: Is Forth code compact? A case study. In: EuroForth 1999 Conference Proceedings, St. Petersburg, Russia (1999)
7. Herzberg, D., Hesenius, M., Reichert, T.: Hands on programming. In: Proceedings of eLBa 2012 (2012) (to be published)
8. Herzberg, D., Reichert, T.: Concatenative programming: An overlooked paradigm in functional programming. In: Proceedings of ICSOFT 2009 (2009)
9. Jung, J.: The research of mobile user interface design components from the standpoint of universal design for learning. In: Proceedings of the IEEE International Workshop on Wireless and Mobile Technologies in Education, pp. 254–256. IEEE Computer Society, Washington, DC (2005)
10. McDirmid, S.: Coding at the speed of touch. In: Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, ONWARD 2011, pp. 61–76. ACM, New York (2011)
11. Pestov, S., Ehrenberg, D., Groff, J.: Factor: a dynamic stack-based programming language. In: DLS 2010 Proceedings of the 6th Symposium on Dynamic Languages (2010)
12. Tillmann, N., Moskal, M., de Halleux, J., Fahndrich, M.: Touchdevelop: programming cloud-connected mobile devices via touchscreen. In: Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. ACM, New York (2011)
13. Xu, S., Bradburn, K.: Usability Issues in Introducing Capacitive Interaction into Mobile Navigation. In: Salvendy, G., Smith, M.J. (eds.) HCII 2011, Part II. LNCS, vol. 6772, pp. 430–439. Springer, Heidelberg (2011)

Domain-Specific Languages in Few Steps
The Neverlang Approach

Walter Cazzola

Department of Informatics and Communication
Università degli studi di Milano
cazzola@dico.unimi.it

Abstract. Often an *ad hoc* programming language integrating features from different programming languages and paradigms represents the best choice to express a concise and clean solution to a problem. But, developing a programming language is not an easy task and this often discourages from developing your problem-oriented or *domain-specific* language. To foster DSL development and to favor clean and concise problem-oriented solutions we developed Neverlang.

The Neverlang framework provides a mechanism to build custom programming languages up from *features* coming from different languages. The composability and flexibility provided by Neverlang permit to develop a new programming language by simply composing features from previously developed languages and reusing the corresponding support code (parsers, code generators, ...).

In this work, we explore the Neverlang framework and try out its benefits in a case study that merges functional programming *à la* Python with coordination for distributed programming as in Linda.

Keywords: Development Tools, Language Design and Implementation, DSL, Composability, Modularity and Reusability.

1 Introduction

Nowadays, several and widely used programming languages support different programming paradigms, such as Erlang [25], Python [19] and Scala [21]. Such a design choice is an attempt to remedy to the lack of conciseness that is often manifest in a traditional general-purpose language. To have at disposal only a programming paradigm is too rigid and forces to write code that awkwardly solve specific problems, e.g., try to imagine how should be to write generic sorting algorithms without first-order functions, function objects or templates.

Even if multi-paradigm programming languages put at disposal several programming paradigms, the different programming models might not offer a concise way to express the desired solutions and often their complexity could be excessive with respect to the requirements. Moreover to let coexist different programming paradigms means to compromise some of the functionality of one or the other paradigm and such a compromise could endanger the expected benefits. Last but not least you are still framed in the language designer's design and what you need could be still missing.

T. Gschwind et al. (Eds.): SC 2012, LNCS 7306, pp. 162–177, 2012.

Another approach to favor the conciseness of the written program is based on extending a programming language via an external library to fulfill the desired missing features (e.g., the ODBC library). In general, this approach is tailored on the desired feature and so cleaner but the adopted syntax and the integration of the introduced feature with the rest of the programming language are far from optimal and sometimes its usage could result cumbersome. Moreover, the API provide a more complex interface if compared with the same feature supported by the language [17] (e.g., compare iterating on a collection in Java with or without the *foreach* construct) and in statically typed languages its integration cannot be as seamless as desired (e.g., some casts could be necessary). To clear up these issues look at the advantages of LINQ (a DSL for DB connectivity) over ODBC based approaches [16].

Other minor issues that should foster the provision of *ad hoc* programming languages are related to efficiency and extensibility. A general-purpose programming language usually provides several programming features but some of them might be unnecessary or redundant (e.g., Python's map/filter/reduce and list comprehensions) and contribute to make the language over complicate to learn and use. Mainstream programming languages have a poor support for extensibility [3, 7] and those that are designed to be extensible (e.g., Lisp, Scala) did not gain a wide acceptance [2] or are really inefficient (e.g., parser combinators in Scala[1]).

The ideal solution to get conciseness, efficiency and extensibility would be to develop a *domain specific* programming language by combining only those features really necessary to solve the target problem, reusing the definition and implementation from other programming languages. This would allow different paradigms to be freely mixed in a fine grained manner and to speed up the design and implementation of a new *ad hoc* programming language. To this respect we have developed the Neverlang [5,6] framework that permits to design new languages in terms of features of other programming languages and to fast generate an interpreter/compiler for such language by reusing pieces of the compilers/interpreters implementing such features.

The rest of the paper has the following organization. Sect. 2 introduces the Neverlang framework whereas some details on the implementation are in Sect. 4. Sect. 3 explores the potential of Neverlang by showing a case study focused on the creation of a concurrent/functional programming language; in particular it shows how to mix two language definitions to create a new language and how to change the behavior of an existing language. Sect. 5 provides a critical analysis of some related works pointing out the differences and the innovations introduced by the Neverlang framework. Last but not least, in Sect. 6 we draw our conclusions.

2 The Neverlang Framework

The Neverlang [5,6] framework is inspired by HyperJ's [23] multi-dimensional separation of concerns and basically reflects the fact that programming languages have a modular definition and each language feature can be easily added to or removed from

[1] http://scala-programming-language.1934581.n4.nabble.com/Performance-of-Scala-s-parser-combinators-td3165648.html

the language. Ideally, the design of a programming language should consist of selecting a set of features (building blocks) from existing languages and composing them together. The whole structure of the compiler/interpreter is the result of composing the code necessary to compile/interpret each single feature. The Neverlang framework realizes this vision and provides a language for writing the building blocks and a mechanism for composing the blocks together and generating the compiler/interpreter of the resulting language.

2.1 Neverlang at a Glance

In the next we describe the basic elements and concepts introduced by Neverlang and the composition model behind the approach.

Basic Framework Concepts. In our approach we exploit the vision that a programming language is defined in terms of its features (e.g., types, statements, relationships, and so on) and such features can be formally described in isolation (as productions of a grammar) and composed to form the language structure (syntax). Traditional compiling techniques [1] perform some transformation on such description that brings forth to the interpretable/compiled code. A complete compiler/interpreter built up with Neverlang is the result of a compositional process involving several basic units describing the language features and their support code.

These basic units are called *module*s. Each module encapsulates a specific feature, such as the syntactical aspect of a loop, and thus it is bound to a precise *role*, the syntax definition in our example. The role category determines where a module will be attached to. Roles can be interpreted as *dimensions* and are bound to the phase of the compilation and interpretation process. `syntax`, `type-checking` and `evaluation` are examples of key roles but they are not the only and fixed set of roles: the user can define new roles (in the language definition) associated with specific compilation phases and he can define the whole compilation process in terms of the defined and included compilation phases giving their relative order, i.e., specifying what phase precedes what (through the keyword **`roles`**).

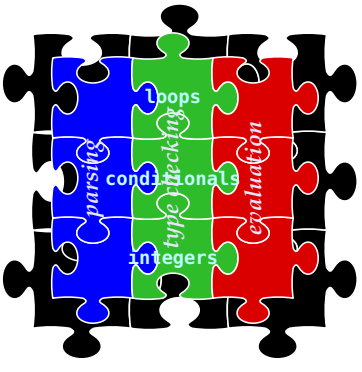


Fig. 1. Sectional DSL

Finally, modules regarding the same language structure but with different roles are grouped together in *slice*s. The final language is simply the result of the slice composition. To some extent, we can say that slices are *orthogonal* to roles: the former are a collection of modules that compose the same feature, the latter are a collection of modules regarding the same compilation/interpretation phase. In this scenario designing a domain specific language consists of defining a set of slices and composing them together. In Fig. 1 is depicted the general multi-dimensional structure of a language developed by using Neverlang, the colored jigsaw pieces are modules, those in the same row contribute to describe the same feature and are part of the same slice whereas their

color specifies which role they play. Neither the number of modules composing each slice must be always the same nor a module for each role must be present in a slice.

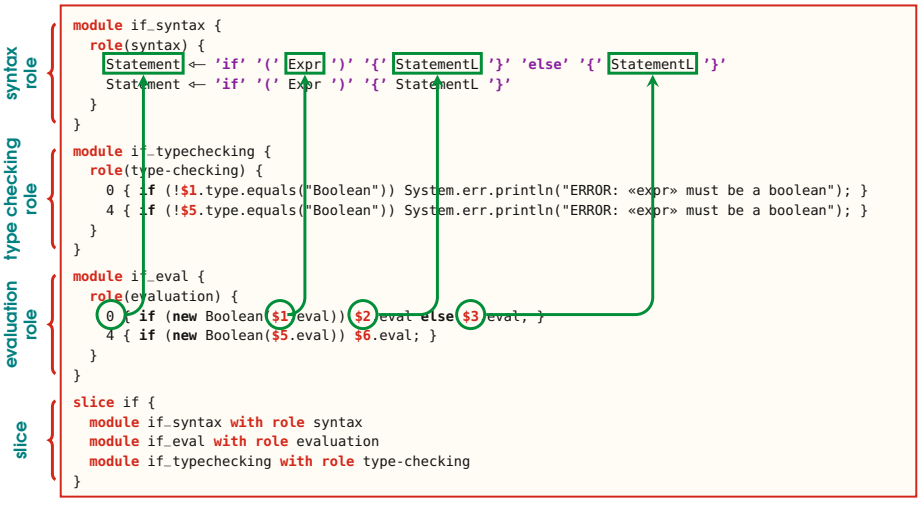
To plug a slice in, we need a mechanism to precisely select the insertion point inside the compiler/interpreter. The process for selecting these insertion points, or join points in aspect-oriented parlance, is grammar driven: they correspond to the nonterminal symbols of the grammar; a grammar that dynamically grows as new slices are plugged in. In our case the code to be introduced at the join points, advice in aspect-oriented jargon, participates to define/implement the compiler/interpreter of the new language and consists of the grammar productions (in the syntactic module) with the related semantic action routines (in the other modules). The Neverlang approach to compiler/interpreter building is *symmetric* [13]. The DSL is a sort of patchwork of only those features selected to be part of the language and its implementation is not achieved by modifying an existing compiler/interpreter but built up from the implementation of the single feature provided by the corresponding slices; that is, the compiler/interpreter is the result of the slice composition. The composition specification defines the grammar join points and its advice. A complete compiler/interpreter reifies its grammar join points, so that it can be subsequently extended with new productions. A pleasant effect of symmetric composition is that many slices can be easily reusable by different DSLs.

To support the various compilation/interpretation phases, the developer may need some ancillary structures or services that concerns the whole compilation process affecting all the other modules crosswise. Simple examples are the *symbol table* and the code to deal with the memory management. A slightly different form of slice called *endemic* supports this kind of behavior. The fields and methods defined in an endemic slice are accessible by all modules in the Neverlang program independently of the compiling/interpreting phase. Adding/replacing an endemic slice permits to easily redefine the whole behavior of the compiler/interpreter (more on this in Sect. 3.4).

Modules and Slices Definition. Listing 1 shows the Neverlang implementation for the *if-else* conditional construct. Three roles (`syntax`, `type-checking` and `evaluation`) are involved; each role is defined in a separate module (that could be written in separate files as well) and combined together in the `if` slice.

Syntactically, the *if-else* construct can be defined by two productions: the first defines the complete case with both branches and the second defines the case without the `else` branch (see [1]). Such productions are defined in a module with the role `syntax`. Each production is composed by *terminal*s (surrounded by `'`) and *nonterminal*s (e.g., `StatementL`). These productions are bound to the nonterminal `Statement`, other productions bound to such nonterminal can be defined in the `syntax` module of other slices. Obviously nonterminals used in the right side of the productions and unbound in the module (such as `StatementL` and `Expr`) must become bound in the slice composition phase where other slices with productions bound to them come into play; otherwise the grammar will be incomplete and the compiler/interpreter could not be generated.

All the other kind of modules (in our example those with role `type-checking` and `evaluation`) add *semantic action*s to the grammar rules defined in the module with `syntax` role (see *syntax-directed translation* in [1]). Each of these modules associates to the nonterminals the semantic actions necessary to carry out the corresponding compilation/interpretation phase — e.g., the semantic actions in the module with role



Listing 1. Simple `if-else` module and slice

`type-checking` are used during the type checking phase. The nonterminals are identified through their position in the productions numbering with 0 the top leftmost nonterminal and incrementing by one left-to-right and up-to-down all nonterminals independently from any repetition and for the whole set of productions defined in the slice. In our example, the module with role `evaluation` defines two semantic actions. The first enriches the head of the first production (position 0) and simply tests the evaluation of the boolean expression associated to the nonterminal in position 1 (`Expr`) and, accordingly to that, respectively evaluates the nonterminal in position 2 or 3 through their `eval` attribute (such associations are made explicit by arrows in the listing). The second action behaves similarly but refers to the case without the `else` branch and it is associated to the head of the second production (position 4). Semantic actions are anchored to a nonterminal; if it is the head of the production, the action evaluation is carried out after the evaluation of the semantic actions in the right part of the production and in its derivation (*postfix* evaluation); otherwise it is done before (*prefix* evaluation).

The semantic actions are basically pieces of Java code that access attributes computed during the current (or previous) compilation/interpretation phases. What the attributes are and how they are transmitted from a module to another derives directly from how the *syntax-directed translation* mechanism [1] works. Attributes are accessed through the nonterminal (by its position prefixed by `$`) they refer to, e.g., `$1.eval` is the `eval` attribute of the `Expr` nonterminal in the first production of our example. To make clear how the slice composition takes place, we can make a parallel with aspect-oriented parlance and consider a nonterminal as a sort of join point where productions and semantic actions are woven during the generation of the compiler/interpreter for the language. Finally, the keyword **`slice`** permits to select the modules that will compose our slice and to specify which role such modules play in the slice. Note that a role can be played by only one module in each slice.

```
# the master process defines the tasks for the slaves and collects the results.
process main { # distribute the jobs to the slaves and collect the results
  rLenA = args[0]              # A's and B's dimensions are passed as args
  cLenA = args[1] rLenB = args[2] cLenB = args[3]

  # A and B are randomly generated.
  A = [[random(10) for y in range(cLenA)] for x in range(rLenA)]
  B = [[random(10) for y in range(cLenB)] for x in range(rLenB)]

  # calculates B^T in a list, then converts it back to a matrix
  B = [B[y][x] for x in range(cLenB) for y in range(rLenB)]
  B = [[B[x] for x in range(y*rLenB, rLenB+y*rLenB)] for y in range(cLenB)]

  # drops the tuples <A_i, B_i^T>, x and y help in retrieving the results.
  [out("data", A[x], B[y], x, y)
     for x in range(rLenA)
       for y in range(cLenB)
  ]
  out("tuples", cLenA*rLenB)
  cnt=rLenA*cLenB

  # create an empty matrix for the result
  C = [[0*y for y in range(cLenB)] for x in range(rLenA)]

  tot=0                                          # collecting the results
  [C[x][y] = tot for i in range(cnt) if in("result", ?x, ?y, ?tot)]
  print(C)
}

# client (slave) calculates and drops in the tuple space the multiplications
process client {
  tot = 0  len = 0
  do {                          # search for non calculated couple of rows
    in("tuples", ?len)
    if (len != 0) {
      out("tuples", len-1)
      in("data", ?A, ?B, ?x, ?y)
      [tot = tot + z for z  in
          [A[i]*B[j] for i in range(len(A)) for j in range(len(B)) ]
      out("result", x, y, tot)
    }
  } while(len == 0)
}
```

Listing 2. Cooperative matrix calculation in the Linda+Python language

3 Neverlang at Work

In the next we present a case study that shows how Neverlang eases to mix up features from different programming languages to form a new one.

3.1 Case Study: Linda+Python

Nowadays multi-core computers are on the rise and the opportunity to program them as a parallel computer and possibly to communicate through their shared memory is coming into the limelight. Some decades ago, Gelernter *et al.* [4,11] introduced Linda, a parallel programming model based on shared memory, called the *tuple space*, to support inter-process communication. The tuple space approach could become topical again in the context of multi-core programming.

Linda is a coordination language with a very limited set of concepts (only six primitives: **`in`**, **`inp`**, **`rd`**, **`rdp`**, **`out`** and **`eval`**) that needs to be embedded in a Turing complete programming language to be useful and usable. Normally, Linda is integrated into

another programming language either by modifying the existing compiler for the host language or by using an external library (as in JavaSpaces [10]). As we explained in the introduction both approaches have drawbacks: modifying a compiler is a time-consuming and error-prone task whereas an external API could badly fit in the original programming language and its use could be complicated and cumbersome. Neverlang can be used to overcome these issues by simply supporting the compiler/interpreter generation for a domain specific programming language. In particular, the coordination language Linda is merged to the functional characteristics of Python (in particular Python's *list comprehension*s) [18] to form the so-called Linda+Python programming language. This is realized by writing down the necessary slices that permit to support (portion of) the two languages by merging them together to define the new language. Of course, we maximize the benefits when the languages to mix up are already implemented in Neverlang and their slices can be reused.

The result of the process is a framework (compiler and interpreter) that permits to compile and to interpret programs written in the Linda+Python idiom. As an example, listing 2 implements a distributed and shared memory-based version of the matrix multiplication algorithm in Linda+Python: keywords from the Linda language are blue-colored whereas the red-colored are Python keywords. The program follows the master/slaves paradigm; two kind of processes are involved: `main` and `client`. The former (master) stores row and column couples in the tuple space and waits for the result. The latter (slave) looks in the tuple space for such kind of couples, multiplies each couple element by element and puts back in the tuple space a tuple with the sum of such products. Note that more than one `client` process can be launched without any conflict to speed up the process.

3.2 Linda+Python Building Blocks

In the long term, i.e., when the Neverlang framework will support enough programming languages, developing the support for the Linda+Python programming language will simply consist of reusing the slices from Python and those from Linda regarding the features of interest and writing some glue code to merge up the whole system. Since we are still far from such a situation we have developed the necessary slices as well.

Linda. The Linda implementation in Neverlang should, at least, provide a slice for each Linda primitive, a slice to support *tuple*s and *anti-tuple*s and the support for the *tuple space*. The slices to implement the primitives and the *tuple*s and *anti-tuple*s do not introduce particularly relevant concepts. The implementation of the *tuple space* is more interesting since it does not introduce any piece of new syntax but it is just a sort of data structure used by the other primitives; it is the perfect example of feature endemic to the rest of the programming language and it is implemented in the endemic slice `TupleSpace` (Listing 3). In this case, the `type-checking` phase does not only support the classic type checking but also the tuple matching to extract tuples from the *tuple space*. As for the *tuple space* management, the `TupleSpace` relies on an external Java package (cf. the `TupleSpaceThread` class in Listing 4). The endemic slice is used by the other slices through the syntax.

```
slice TupleSpace {
  decl {
    import TupleSpace.TupleEntry; import TupleSpace.TupleSpaceThread;
    TupleSpaceThread ts = TupleSpaceThread.getInstance();                    // tuple space
    Hashtable<String,String> getMatches(ArrayList<String> val,ArrayList<String> types,boolean del) {
      TupleEntry res, te = new TupleEntry(types, val);                       // anti-tuple
      if(!del) res = ts.getMatches(te);                                      // read operation
      else res = ts.getMatchesAndRemove(te);                                 // in operation
                                  ...
    }
    void addEntry(ArrayList<String> values, ArrayList<String> types) {
      ts.addEntry(new TupleEntry(types, values););
    }
  }
  module TupleSpace with role endemic
}
```

Listing 3. The endemic slice to support the tuple space

```
public class TupleSpaceThread implements TupleSpaceInterface {
  private static TupleSpace ts;
  public static TupleSpace getInstance() {
    if (ts == null) ts = new TupleSpace();
    return ts;
  }
  public void addEntry(TupleEntry te) { ... }
  public TupleEntry getMatch(TupleEntry te) { ... }
  public TupleEntry getMatchesAndRemove(TupleEntry te) { ... }
}
```

Listing 4. The external `TupleSpaceThread` library

```
«slice name».«operation name»(«arguments»).
```

The endemic slices are always available and do not need to be imported.

Python. In the case of Python, we are just interested in its list datatype and in the comprehension mechanism. Python lists are heterogeneous and dynamically typed. Dynamic typing forces to postpone the list type evaluation to the evaluation phase and in the case of Linda+Python example to have a mixed type checking: static for the Linda part and dynamic otherwise. The Neverlang framework permits to suspend and to resume a phase on part of the AST; this feature enables the system to postpone the type checking at the evaluation phase.

Due to space limitations, we cannot report the whole slice dealing with the list comprehension feature but we are interested in showing how the suspend/resume mechanism works. Listing 5 shows how to support dynamic checking in a `for-each`-like construct that iterates on a heterogeneous list and executes an expression on every element. During the `type-checking` phase, the expression associated to the `SimpleExpression` nonterminal cannot be type checked because it is (presumably) bound to the identifier (described by the `Identifier` nonterminal) whose type is not available before the evaluation and potentially it can change at each loop since Python's lists are not bound to a single type. The `type-checking` phase for the `SimpleExpression` nonterminal must be suspended by using the special function **`$suspend`**. During the `evaluation` phase, the type and value of the elements of the list are stored in a *variable table* that can be used

```
module ForEach {
  role(syntax) {
    ForEach <- SimpleExpression 'for' Identifier 'in' List
  }
  role(type-checking) {
    1 { $suspend; }
        ...
  }
  role(evaluation) {
    0 {
      String listValue = $3.eval;
      String[] values = VarTable.getValues(listValue);
      String[] types = VarTable.getTypes(listValue);
      for(int i=0; i<values.length; i++) {
         VarTable.putValue($2.eval, values[i]);
         VarTable.putType($2.eval, types[i]);
         $1.resume("type-checking");
         $1.eval;
      }
    }
  }
}
```

Listing 5. Postponed type checking in the implementation of the comprehensions

```
module Inc {
 role(syntax) { SimpleExpression <- Identifier '++' }
 role(type-checking) {
   0 {            // type checking resumes here
     if !(VarTable.getType($1.eval).equals("int")) Logger.printError("Invalid type!");
     else $0.type = "int"
   }
 }
 role(evaluation) {
   0 { VarTable.putValue($1.eval, VarTable.getValue($1.eval)+1); }
 }
}
```

Listing 6. Module `Inc` to implement the increment operation

by the type checker when resumed (call to the **`$resume`** special function). Given that the `SimpleExpression` nonterminal expands into the increment operation (defined by the `Inc` module, Listing 6), the type checking process resumes in the semantic action associated to the head in such module, i.e., the place where the phase is resumed depends on the program we are compiling. In the `type-checking` and `evaluation` phases the values stored in the variable table before the **`$resume`** is used; the variable table is implemented by the `VarTable` endemic slice as a hash table.

3.3 Building Linda+Python Up

Once the necessary slices have been developed (or selected if already existing), we have to define (through the **`language`** statement) how such slices are composed together to form the new Linda+Python programming language; such definition will drive the Neverlang framework in the building of a compiler/interpreter for the new language.

Being the compilation phases syntax-driven, most of the issues the developer has still to face are the syntactical conflicts that could rise by composing modules with role

```
language Linda+Python {
  slices Main BoolOp Sum Process Mul Out Read InP ReadP Eval Length Id ListsOp WildCardExpr Boolean
     Int Assign ListsAccess Range In CompVarTable Tuple Args Expr Rand Var VarTable Comprehension
     DoWhile If OpTable MulTable ListAssign TupleSpace IntOp Lists Print BooleanOpTable SumTable
     ListsTable CompFor
  roles syntax < type-checking < evaluation
}
```

Listing 7. Linda+Python Definition

`syntax` defined by different development teams. A quite common example of this problem occurs when you compose statements such as `if` and `do-while`, whose productions have different left-hand nonterminals rather than a common one as `Statement`; such problem requires some glue code to redefine or to make uniform the productions. Neverlang avoids further conflicts in the other phases since the semantic actions are associated to the nonterminal position and this is related to a specific production independently of the slice composition.

Listing 7 shows the language composition in our case study; such a piece of code just lists which slices should be composed and in which order the compiling phases occur (expressed through the keyword **`roles`**).

3.4 Flushing Flexibility Out

The tuple space implementation and the distribution provided in Listing 3 are quite naïve; Linda's processes are implemented as threads and the tuple space resides in the data area common to all threads. Of course, this is just a proof of concept but it permits to show another peculiarity of the Neverlang approach: how easy it is to evolve a programming language (more on DSL maintenance in Neverlang can be read in [5]). Switching from the current thread-based implementation to a more distributed RMI-based one is just a matter of substituting the `TupleSpace` slice with another slice supporting the desired implementation while retaining the interface.

Listing 8 shows the endemic slice `TupleSpaceRMI` that will replace the thread-based implementation of the tuple space. The instance of `TupleSpace` does not reside in the common data area anymore but it is accessed through RMI as a remote object. Other minor changes are not showed due to sake of space but the access methods are *synchronized* and the `Tuple` are *serialized* to be stored into or retrieved from the tuple space.

This kind of change is particularly easy since it just affects the run-time environment of a running program and not its syntax and semantics and (with some care for consistency) it could be done at run-time as well. Of course any kind of language evolution can be easily taken in consideration but often this affects the source code as well due to changes to the language syntax. Note that similar flexibility can be achieved also with library based solutions but with a less clean syntax; moreover it is hard to change or extend features whose implementation is less self-contained and library based. In [5] are shown more elaborated kind of evolutions.

The complete case study can be downloaded from the Neverlang web page:

`http://cazzola.dico.unimi.it/neverlang.html`

```
slice TupleSpaceRMI {                    // TupleSpaceRMI differs from TupleSpace in
          ...                            // the way the tuple space is retrieved
  decl {
    TupleSpace ts = connect();
    TupleSpace connect() {
      try {
        return (TupleSpaceInterface)LocateRegistry.getRegistry(Args.hostname).lookup("TupleSpace");
      } catch (Exception e) { ... }
          ...
    }
    public class TupleSpaceRMI implements TupleSpaceInterface {
                ...
      public static void main(String args[]) {
        try {
          TupleSpace ts = TupleSpace.getInstance();
          TupleSpaceInterface stub = (TupleSpaceInterface)UnicastRemoteObject.exportObject(ts, 0);
          LocateRegistry.getRegistry().rebind("TupleSpace", stub);
        } catch (Exception e) { e.printStackTrace(); }
      }
    }
  }
  module TupleSpaceRMI with role endemic
}
```

Listing 8. RMI-based tuple space implementation

4 Neverlang Close-up

Basically, the idea behind the Neverlang framework is to compose the slices listed in the **`slices`** section of the **`language`** statement and to exploit the syntax-directed translation [1] approach on the context-free grammar that results from the `syntax` module composition which is then decorated with the semantic actions specified in the remaining modules.

The resulting compiler/interpreter is mainly composed of two parts: i) a front-end that parses the source files written in the new language and generates the classes that will compose the *abstract syntax tree* (AST) and the AST itself and ii) a type-driven back-end that attaches (through aspect-oriented programming) the semantic actions specified by the developer to the classes composing the AST and traverses the AST to carry out all the compilation/interpretation phases.

Front-End Generation. To render the language definition extensible and sectional we adopted the *parsing expression grammar*s (PEGs) [9] and the Rats! [12] an extensible parser generator that works on these kind of grammars. PEGs look similar to context free grammars but they are not ambiguous: if a string parses, it has exactly one valid parse tree and it is always possible to write a recursive-descent parser running in linear time (the pakrat parser [12]).

The compiler generation procedure starts by collecting all grammar productions contained in the modules with role `syntax` and by translating them in Rats! modules that define the parser for the new language. The tool builds a class (a sort of empty skeleton to be filled later by the back-end) for each nonterminal of the generated grammar; such classes will be used to instantiate the nodes of the AST accordingly to the grammar of the defined language.

The most natural way of representing an AST is to model the language constructs as a class hierarchy with general abstract classes like `Statement` and `Expression`, and specialized concrete classes like `Assignment` and `AddExpression`. In our case, all nonterminals are modeled as abstract classes and the productions permit to specialize them in concrete subclasses. All nonterminals have a common superclass, called `AbsNode`, containing the fields and methods common to all nonterminals; this is refined in concrete classes representing the nonterminals and containing their own inherited and synthesized attributes [1].

Semantic Back-End Generation. The semantic actions associated to a nonterminal are appropriately injected into the AST node representing such nonterminal. Once all the semantic actions related to every compilation/interpretation phase are injected, the compilation/interpretation process can start. Each phase is associated to a specific `visit()` method and will be carried out during the AST traversal by calling such method on each node.

The evaluation carried out at each node is both type-driven (i.e., associated to the corresponding nonterminal) and context-driven (i.e., related to the semantic action associated to the position of the nonterminal in a given production; a nonterminal can occur in several positions and in each position can be decorated by a different semantic action). Therefore, the code of the method invoked during the visit will change on a per node basis (Polyglot [20] has similar necessities solved by delegation). AspectJ [15] represents the perfect tool to realize the required context-driven adaptation of the AST nodes. The code of the semantic actions associated to each nonterminal is automatically woven into the method invoked during the tree traversal accordingly to the type of the node, to the node position in the AST (and consequently the position the corresponding nonterminal has in the applied production) and to the compilation/interpretation phase (that is, the role we are effectively playing). By using this approach, semantic roles are implemented without modifying the classes of the AST nodes and the whole role can be easily plugged and unplugged. This context-adapting `visit()` method implements a sort of aspect-oriented *modular visitor pattern* [22] that permits to avoid the well-known *expression problem* [27]; with respect to Oliveira's [22] proposal, this has also the benefit of reducing the necessary casts.

In detail, a pool of aspects is created for each role (or compilation/interpretation phase if you prefer). Such aspects wrap up the pieces of code that should be attached to the method called during the AST traversal. Each collection of aspects contains also a special element called *driver aspect* that drives the entire compilation/interpretation phase and the switch from a phase to the following; the mechanism is quite simple: the generated compiler/interpreter has a hook in its main program (a dummy method invoked just after the tree construction, that permits to put in evidence a join point). This hook is used by the driver aspect as an anchor where to hook up the AST `visit()` method for a given compilation/interpretation phase (woven before the method call) and the code to switch to the next phase (woven after the method call). At each phase, the AST traversal depends on a flag attached to each node: if the flag is unmarked the node and its children are skipped during the visit; normally all the nodes are marked for the visit. The **`$suspend`** primitive unchecks the corresponding node for the current phase and a special aspect is created to be used when resuming; the **`$resume`** primitive

represents a join point where such aspect is woven to permit the belated visit of the AST. Methods and fields defined in endemic slices are wrapped in static classes and imported by all the generated aspects and classes. These fields are initialized before the AST traversal starts and the implemented services are available during the whole compilation procedure.

To establish an order among the compilation/interpretation phases as expressed by the `roles` keyword, we exploit the advice precedence feature of AspectJ that permits to specify in which order to apply the advices matching the same join point. In particular, in the `main` procedure we set the order in which the *driver aspect*s are woven into the dummy methods.

5 Related Work

Several works share Neverlang's goals. JastAdd, xText and Polyglot are the most pertaining.

JastAdd. The JastAdd [8, 14] system enables open modular specifications of extensible compiler tools and languages. JastAdd is an extension to Java that supports a specification formalism called rewritable circular reference attributed grammars.

JastAdd and Neverlang share a very similar object-oriented implementation of the AST [8]. Moreover, they both adopt aspect-oriented programming to extend the language behavior by injecting methods and fields in the AST nodes. On the other side, in Neverlang the AST nodes and their connections come after the grammar productions whereas in JastAdd they can be user-defined granting a major flexibility but the generated code can bloat.

JastAdd adopts *reference attributed grammars*, i.e., a semantic action in q can refer to an attribute of an unrelated nonterminal r. PEGs, adopted by Neverlang, do no support this feature but it can be simulated by saving r in an external data structure (through an endemic slice) during the AST visit (as we do to deal with the attributes).

In JastAdd each declared behavior rewrites the AST tree nodes giving the opportunity to add or delay a phase of compilation; behaviors are similar to Neverlang roles. Even if Neverlang's modularity (roles) is not limited to compiler phases but straddles the whole compilation/interpretation process via the endemic slices.

Polyglot. Polyglot [20] is an extensible compiler framework that supports the creation of compilers for Java-like languages. Polyglot relies on an *extensible parser generator* that permits to express the language syntactical extensions as changes to the Java grammar.

Polyglot extensibility is supported by *delegation*. Each compilation phase is supported by a delegate object present in each AST node type; the delegate object is appropriately replaced in each extension.

Neverlang and Polyglot share similar goals, i.e., supporting the development of syntactical and semantical extensions to a programming language but Polyglot is limited to Java. Besides, Polyglot extensions are just source-to-source translations from the extended language to pure Java. Modularity and reusability are issues that Polyglot does not face.

xText. xText is an Eclipse plug-in that provides a framework for the development of domain-specific languages. It is tightly integrated with the Eclipse modeling framework [26] to provide a language-specific IDE.

Like JastAdd the user is free to define the relation between grammar productions and AST nodes but each parser rule will create a new node in the AST. The language meta-model describes the structure of its AST.
xText generator leverages the *modeling workflow engine* from Eclipse modeling framework technology and the code is generated from the meta-model created by the parser; the meta-model is similar to the Neverlang semantic back-end.

The framework gives the opportunity to reuse existing grammars and existing meta-models to implement the back-end for different languages. However the framework seems oriented to infer a model from a text and to translate it to an other model (*model-driven development*) rather than to create real compilers. A similar approach (model-driven) is also provided by Frag [28].

To recap, the main difference, that evinces from this comparison, is that Neverlang focuses on modularity and reusability of the compiler units; in Neverlang the developer can easily extend and mix existing languages to define new languages with working compilers/interpreters. Moreover, by compiling Neverlang programs we get real compilers/interpreters for programming languages completely independent of the language used to implement the compiler (Java in our case) and not (source-to-source, model-to-model, ...) translators that limit the implemented programming language to syntactic extensions of the host language.

6 Conclusions

In this paper we have introduced Neverlang: a framework to describe new programming languages as the composition of programming features from existing programming languages and to generate the compiler/interpreter for the new language by reusing previous implementations. Moreover we have shown how Neverlang can be used to mix up programming features from Python (*list comprehensions*) and Linda (*coordination*) to form a multi-paradigm programming language and showed how its implementation can be easily changed from a thread-based tuple space to a RMI-based one.

Currently an incremental parser (similar to the PetitParser [24]) is under development, the idea is to drop the PEG-based parser in favor of a more flexible parser that will permit to avoid of regenerating the whole parser when a slice is added or removed from the language. This would permit to change the behavior of a running system, for example, in the Linda+Python case study we could change the tuple space implementation from one version to another on-the-fly. Also the implementation of some well known languages like Java and Python is under development as well as the improvement of the composition mechanism to support a finer decomposition for the feature implementation and to ease the mix up of different interpretation/compilation philosophies (e.g., monadic and traditional interpretation).

Acknowledgments. The author wishes to thanks Ivan Speziale and Davide Poletti that worked on the Neverlang implementation; without their help Neverlang would be a nice *empty* box.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison Wesley, Reading (1986)
2. Batory, D., Lofaso, B., Smaragdakis, Y.: JTS: Tools for Implementing Domain-Specific Languages. In: Proceedings of the 5th International Conference on Software Reuse, Victoria, BC, Canada, pp. 143–153. IEEE Computer Society (June 1998)
3. Bravenboer, M., Visser, E.: Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation without Restrictions. In: Vlissides, J.M., Schmidt, D.C. (eds.) Proceedings of the 19th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004), Vancouver, BC, Canada, pp. 365–383. ACM (October 2004)
4. Carriero, N., Gelernter, D.: Linda in Context. Commun. ACM 32(4), 444–458 (1989)
5. Cazzola, W., Poletti, D.: DSL Evolution through Composition. In: Proceedings of the 7th ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution, RAM-SE 2010, Maribor, Slovenia. ACM (June 2010)
6. Cazzola, W., Speziale, I.: Sectional Domain Specific Languages. In: Proceedings of the 4th Domain Specific Aspect-Oriented Languages, DSAL 2009, Charlottesville, Virginia, USA, pp. 11–14. ACM (March 2009)
7. Ekman, T.: Extensible Compiler Construction. Phd thesis, Department of Computer Science, Lund University, Lund, Sweden (2006)
8. Ekman, T., Hedin, G.: The JastAdd Extensible Java Compiler. In: Proceedings of the 22nd Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 2007, Montréal, Québec, Canada, pp. 1–18. ACM (October 2007)
9. Ford, B.: Parsing Expression Grammars: a Recognition-Based Syntactic Foundation. In: Jones, N.D., Leroy, X. (eds.) Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, pp. 111–122. ACM (January 2004)
10. Freeman, E., Arnold, K., Hupfer, S.: JavaSpaces Principles, Patterns, and Practice. Addison-Wesley (1999)
11. Gelernter, D.: Generative Communication in Linda. ACM Trans. Prog. Lang. Syst. 7(1), 80–112 (1985)
12. Grimm, R.: Better Extensibility through Modular Syntax. In: Schwartzbach, M.I., Ball, T. (eds.) Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, PLDI 2006, Ottawa, Ontario, Canada (June 2006)
13. Harrison, W., Ossher, H., Tarr, P.: Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition. Technical Report RC22685 (W0212-147), IBM (December 2002)
14. Hedin, G., Magnusson, E.: JastAdd — An Aspect-Oriented Compiler Construction System. Science of Computer Programming 47(1), 37–58 (2003)
15. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Lee, S.H. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
16. Marguerie, F., Eichert, S., Wooley, J.: Introducing LINQ. In: LINQ in Action. Manning (January 2008)
17. Mernik, M., Heering, J., Sloane, A.M.: When and How to Develop Domain Specific Languages. ACM Comput. Surv. 37(4), 316–344 (2005)
18. Mertz, D.: Functional Programming in Python. In: Charming Python, ch. 13 (January 2001), http://gnosis.cx/publish/programming/charming_python_13.txt
19. Nielson, S.J., Knutson, C.D.: OO++: Exploring the Multiparadigm Shift. In: Prooceedings of ECOOP Workshop on Multiparadigm Programming with Object-Oriented Languages, MPOOL 2004, Oslo, Norway (June 2004)

20. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An Extensible Compiler Framework for Java. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 138–152. Springer, Heidelberg (2003)
21. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Aritma Press (2008)
22. Oliveira, B.C.d.S.: Modular Visitor Components: A Practical Solution to the Expression Families Problem. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 269–293. Springer, Heidelberg (2009)
23. Ossher, H., Tarr, P.:Hyper/J: Multi-Dimensional Separation of Concerns for Java. In: Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, Toronto, Ontario, Canada, pp. 729–730. IEEE Computer Society (2001)
24. Renggli, L., Ducasse, S., Gîrba, T., Nierstrasz, O.: Practical Dynamic Grammars for Dynamic Languages. In: Proceedings of the 4th Workshop on Dynamic Languages and Applications, DYLA 2010, Málaga, Spain (June 2010)
25. Sahlin, D.: The Concurrent Functional Programming Language Erlang – An Overview. In: Proceedings of the Joint International Conference and Symposium on Logic Programming, Bonn, Germany (September 1996)
26. Steinberg, D., Budinsky, D., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. Addison-Wesley (December 2008)
27. Torgersen, M.: The Expression Problem Revisited. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 123–146. Springer, Heidelberg (2004)
28. Zdun, U.: A DSL Toolkit for Deferring Architectural Decisions in DSL-Based Software Design. Information and Software Technology 52(7), 733–748 (2010)

Business Process Lines and Decision Tables Driving Flexibility by Selection

Nicola Boffoli, Danilo Caivano, Daniela Castelluccia, and Giuseppe Visaggio

Department of Informatics- University of Bari- Via E. Orabona 4- 70126-Bari-Italy
{boffoli,caivano,castelluccia,visaggio}@di.uniba.it

Abstract. A major challenge faced by organizations is to better capture business strategies into products and services at an ever-increasing pace as the business environment constantly evolves. We propose a novel methodology base on a Business Process Line (BPL) engineering approach to inject flexibility into process modeling phase and promote reuse and flexibility by selection. Moreover we suggest a decision-table (DT) formalism for eliciting, tracking and managing the relationships among business needs, environmental changes and process tasks. In a real case study we practiced the proposed methodology by leveraging the synergy of feature models, variability mechanisms and decision tables. The application of DT-based BPL engineering approach proves that the Business Process Line benefits from fundamental concepts like composition, reusability and adaptability and satisfies the requirements for process definition flexibility.

Keywords: business process management, business process modeling, business process line, feature model, variability mechanisms, decision table.

1 Introduction

Business processes and services are at the heart of an ongoing "silent revolution". A major challenge faced by organizations in today's environment is to monitor a constant evolution of business environment and better capture business strategies into products and services at an ever-increasing pace as the business environment evolves. At the same time, organizations distributed by space, time and capabilities are increasingly pushed to exploit synergies by integrating their business processes in order to produce new value-added products and services. Also mergers or acquisitions can entail integration of different processes, reuse of parts of the process to be discarded, inclusion of parts of other processes and so on.

Both the process evolution due to internal and external factors, the Business Process Management (BPM) paradigm stresses the importance of integrating whole process rather than simply integrate data or applications [1, 2]. Also, the *process wave* initiated by Hammer and Champy [3] led to the awareness of business process models as indispensable artifacts to drive business management and evolution. However, traditionally, BPM systems were used to support static business processes, in sense of processes which do not change frequently. This has limited the scope of this

T. Gschwind et al. (Eds.): SC 2012, LNCS 7306, pp. 178–193, 2012.

management. Business process modeling management systems and languages that are able to describe and unroll dynamically changing processes are today necessary.

In this scenario, in our research work, we face how to allow a business process to evolve in an agile manner by injecting flexibility into process definition, so we approach the problem of flexibility from a modeling perspective. Flexibility is the adaptation to a changing environment. However, adapting process models or its instances during their execution fit only stable processes in expecting changes domain. In order to provide the capacity to anticipate the change at modeling-time, we argue that a more systemic view of business process variability is necessary to handle the problem 'in the large' and we suggest capturing changes in a product-line engineering approach based on selection and design of commonalities and variations. Seeing the duality that exists between products and processes, we believe that lines of business processes could beneficially be handled as software product lines: we propose to model business processes in a *Business Process Line* (BPL) able to elicit commonalities as well as variant components in order to capture process variability and promote reuse and flexibility needed in a constantly changing business environment.

For this purpose, after a feature-oriented domain analysis as typically in software product line development, our proposed methodology intends to inject appropriate variability mechanisms in process modeling in order to enable flexibility by selection. Variability mechanisms [4] takes inspiration from modularity concept in the object oriented paradigm to establish a hierarchical construction of the business process modeling: so it can be possible to model business processes with inheritances, encapsulations, extensions, parameterizations and so on. The resultant business process model will benefit in terms of flexibility by leveraging of fundamental concepts like composition, reusability and adaptability and offer ease of change to analysts.

However, in order to anticipate the process changes, it is essential to elicit all possible process characteristics and not to leave degrees of freedom in process execution, so we suggest to track and manage the considerable amount of process parameters through a decision-oriented paradigm: we introduce the use of *decision tables* (DTs) to acquire, formalize and reuse all the emerging decision points during business process modeling. Decision tables are able to give a representation of the relationships among business needs, environmental changes and process tasks in a complete manner, without inconsistencies. These peculiarities are assured by compact overview of a large number of information, modular knowledge organization, effective verification of consistency, completeness and redundancy. Moreover, decision tables are easily maintainable by supporting the dynamic reengineering of the represented relationships. This paradigm seems to be particularly appropriate for representing knowledge intensive business processes or any kind of processes requiring flexibility.

Briefly, our research work investigates the following Research Questions (RQ):

- RQ1: How to capture and model variability in business processes in order to improve flexibility in changing environment?
- RQ2: How to elicit and manage all possible process characteristics to prevent and detect anomalies in decision points?

To face this research questions, the following solutions are respectively proposed: business-process-line engineering approach and decision-table-based support.

The structure of this paper is the following one. Section 2 introduces the proposed methodology. This section explains what a Business Process Line is and how to build it in a real case study. The section 3 describes the support of decision table formalism. The section 4 discusses the related works and finally the section 5 draws the conclusion.

2 Business Process Lines Boosting Process Flexibility

Process flexibility is the capacity of making compromise between, first, satisfying rapidly and easily the business requirements in terms of process adaptability when organizational, functional and/or operational changes occur; and second, keeping effectiveness [5]. The issue of business process flexibility can be addressed from two great perspective [6], as shown in figure 1:

- *Flexibility by Selection (A Priori).* It is based on modeling formalisms which offer the capacity of taking into account the environmental changes without changing the definition of the business process. It ensures the existence of a number of alternatives of execution in the business description at design time. Decision points have to be perfectly represented. It is recommended in the case of process for which we can know in advance all the possible execution cases. Nevertheless, users note that there are processes for which they cannot always anticipate all the possibilities of execution at the design time.
- *Flexibility by Adaptation (A Posteriori).* It adapts the definition of the business process without anticipating the capacity of change of the process at its design time. It injects flexibility at execution time by means of a configuration mechanism in order to ensure the change of parameters, the change of the execution way, or the addition of new participants.

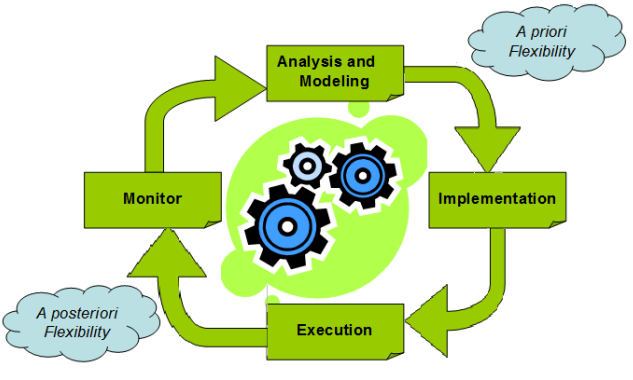


Fig. 1. Business Process Flexibility Schema

Many approaches have been proposed to address the issue of flexibility support in business process modeling and enactment. In this paper we focus on *flexibility by selection* at build-time in order to leverage the capacity to anticipate the changes by means of the proposed process modeling approach. In fact, the BPL approach, as well as in software product line engineering, aims at exploiting typical benefits from modularity and component composition able to add flexibility at modeling and design time according to ever-changing environments.

The situation is similar to that in manufacturing and product engineering where the notions of product lines have been introduced [7]. Design of product lines demonstrated the need to elicit commonalities as well as variable parts in a product and stressed the importance of the variability concept. Managing commonalities and variability leads to two major advantages: reuse of common parts [8], [9] and adaptation to different customers and various organizational settings [10].

Moreover numerous reports document the significant achievements gained by introducing software product lines in the software industry [11] in terms of software engineering flexibility.

2.1 What Is BPL

According to the Software Product Line paradigm, our proposed methodology is intended to orchestrate a Business Process Line in order to manage a set of similar business processes as follows:

1. share common assets (commonality) among all the business processes;
2. characterize each business process by one or more variant assets (variability) depending on the specific context where the process will be applied;
3. introduce flexibility by selection (*a priori*) in business processes in terms of features they provide, the requirements they fulfill.

A Business Process Line is a portfolio of closely related processes with variations in features and operative contexts, rather than just a single business process. A BPL consists of:

1. a set of *invariant assets* (commonality): they are common process parts which constitute the basis for all processes of BPL;
2. a set of *variant assets* (variability): they are specific process parts related to a subset of features which can be selected to tailor the target process; variant assets must be designed with the most appropriate level of granularity by encapsulating the atomic process parts and hiding the problem-solution relationships in order to provide strategic, large-grained reuse;
3. a set of *rules*: they explicit the decision-making task to perform flexible composition of process assets (variant and invariant ones).

2.2 How to Build a BPL

A) Domain Analysis. Feature models were first introduced in the Feature-Oriented Domain Analysis (FODA) method by Kang [12]. Since then, feature modeling has

been widely adopted by the software product line community. Feature models capture the functional and non-functional requirements of the products in the software product line and the decisions about common and variant capabilities and behaviors across the product line.

According to FODA, a feature is "a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems". We borrow this definition for BPL and define a *business feature* as "a representation of a visible process characteristic and an abstraction of a cohesive business flow of activities". Therefore we use feature models in order to represent features and their dependencies in business scenario, typically in the form of a feature diagram with cross-tree constraints (figure 2).

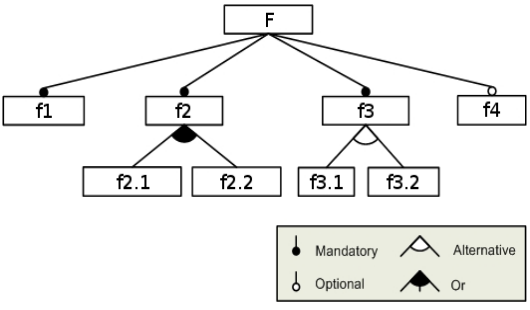
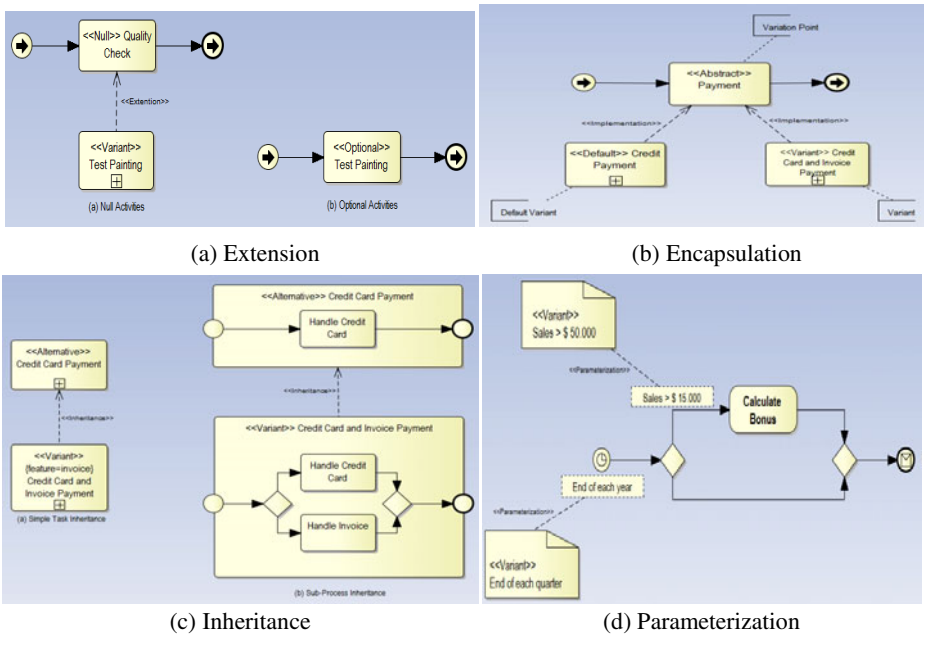


Fig. 2. An example of Feature Tree notation

B) Modeling Variability. Variability mechanisms denote techniques for the derivation of process model variants from existing process models. We introduce them in our BPL approach in order to inject variability in the business process model (samples in figure 3). Similarly to the Object Oriented paradigm, the definition of variability mechanisms refers to the following categories (details in [13]):

1. *Extension.* Extensions and extension points are used to extend an encapsulated process asset at predefined points, the extension points, by additional optional behavior selected from a set of possible variants. An extension point activity is marked with the stereotype <<Null>>, associations marked with <<Extension>> connect optional implementations.
2. *Encapsulation.* Specific process assets are inserted into an invariant interface. Thereby, an interface is defined as the set of input and output events of an activity. The interface activity is marked with the stereotype <<Abstract>>. Possible realizations of the interface are connected using associations marked with <<Implementation>>.
3. *Inheritance.* Inheritance adds restrictions to addition/omission/replacement of single elements. Inheritance modifies an existing (default) process asset by adding activities regarding to specific rules. An association represents inheritance from the child activity to the parent activity when it is marked with the stereotype <<Inheritance>>.

4. *Parameterization*. Using parameterization processes are built by a generic process with a set of parameter values. In BPMN, each attribute can be parameterized to support optional, alternative, or range variation points.



(a) Extension (b) Encapsulation

(c) Inheritance (d) Parameterization

Fig. 3. Variability Mechanisms Samples

2.3 Illustrative Case Study

In order to motivate the discussion about modeling for flexibility with BPL engineering approach, let us consider the following scenario. In South Italy companies are working on developing a selling system for local agricultural and food market. They are collaborating with our research team in order to implement their business processes for different customers and automate them via web services. Within this project there are different providers offering a number of selling services that could be reused and adapted in each different business contexts according to the customer needs. That's why the project represents a field of interest for the proposed approach application.

Starting from the analysis of MIT library [14] we have achieved information useful to obtain a BPL library. The MIT library represents a collection of more than 5000 business activities related in several business processes. In particular, we have extracted the set of business activities needed for building our Selling BPL: *Share out goods, Register Sellers, Register Alternative Products, Arrange store displays, Auction, Check quality, Register Auction Result, Identify potential customers need,*

Identify potential customers, Inform potential customers, Manage customer relationships. For space reason, in this section we show the BPL subsystem concerning the "Auction" business activity. In figure 4 there is the initial BPMN model of business process (open ascending-price auction).

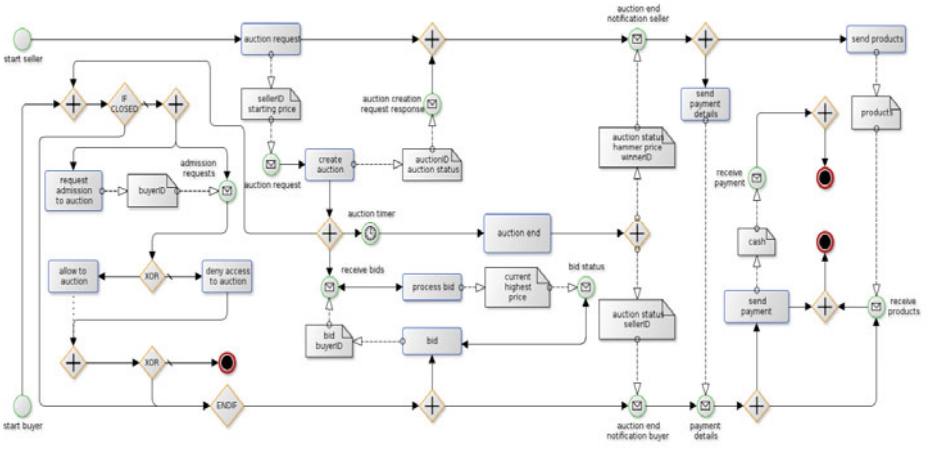


Fig. 4. Initial Process "Auction"

A) Feature Model. Firstly, we analyzed Auction business scenario searching for peculiar features. As shown in figure 5, the resultant Auction feature model consists of the following elements, some of them mutually exclusive:

- "open", auction accessible to all e-marketplaces; "closed", auction accessible to invited sellers/buyers;
- "direct", bids from the seller to buyers; "inverse", bids from the buyer to sellers;
- "english", ascending price auction; "dutch", descending price auction; "hybrid", the auction switches from english mode to dutch one if there isn't a starting price appropriate for the seller.

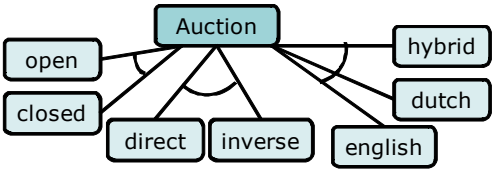


Fig. 5. Auction Feature Model

Starting from the feature model, for each feature we conducted the *variability analysis* step in order to identify the invariant part of the auction process and a set of variant process elements (*Variant Assets*). For example, Open/closed feature analysis lead to identify "Buyer Login" variant asset, according to the optional requirement of login for the invited buyers.

B) Variability Mechanisms. In order to formally represent the auction process, we modeled the process in BPMN formalism, starting from the invariant asset and adding the variant assets by means of the variability mechanisms. Based on the feature tree, we detected and modeled the variant and invariant process assets by means of the variability mechanisms. We injected the following variant assets into the process model:

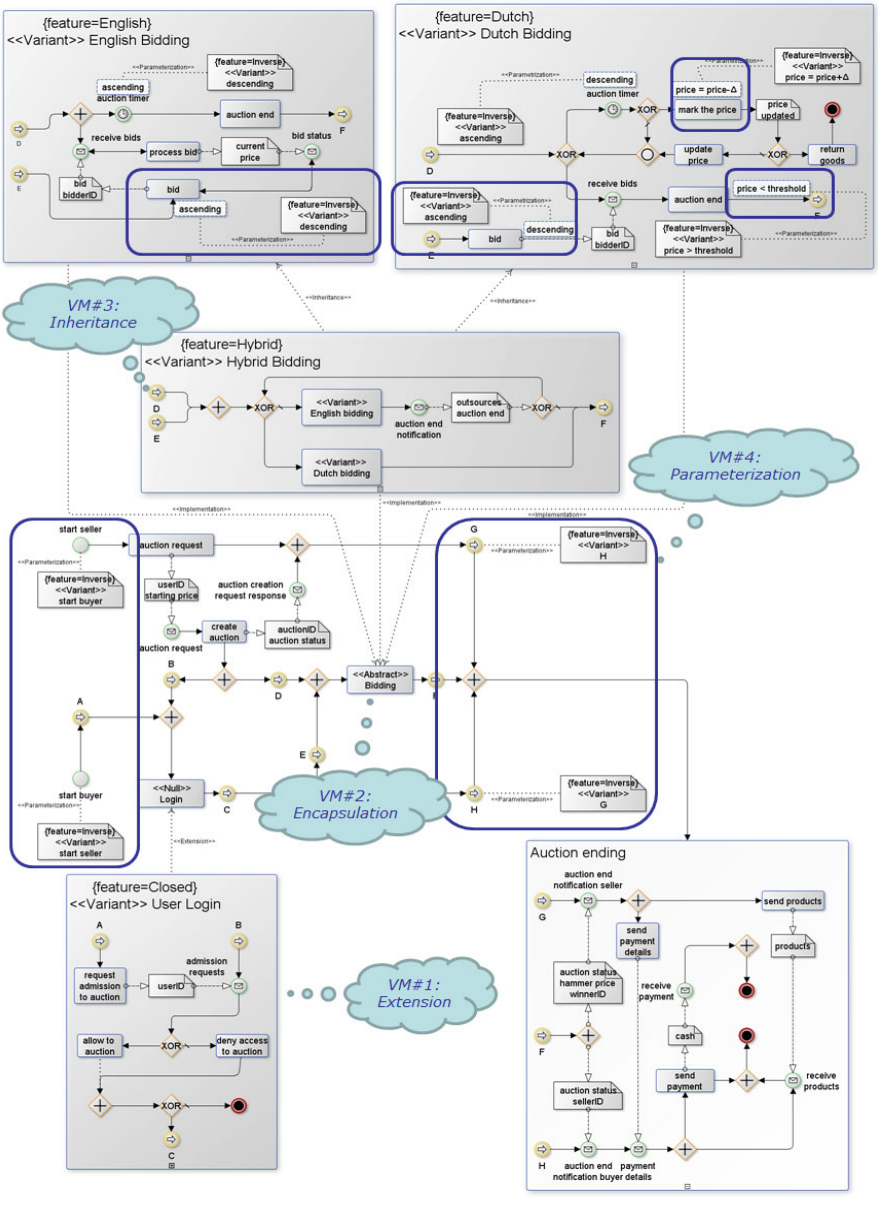


Fig. 6. Variability Mechanisms in Auction process model

- "Buyer login" variant asset is connected to the invariant auction preliminary steps, through the *Extension* variability mechanism;
- "English bidding" and "Dutch bidding" variant assets are connected to the abstract bidding task through the *Encapsulation* variability mechanism;
- "Hybrid bidding" variant asset is connected to "English bidding" and "Dutch bidding" assets through the *Inheritance* variability mechanism.

Finally, we injected the *Parameterization* variability mechanism in order to generalize some tasks (i.e. "bid", "mark the price", "auction end") into the process model, by means of specific parameters that enable "Direct" and "Inverse" features. Figure 6 shows the process model after the injection of all the variability mechanisms.

2.4 The Validation Issue

By leveraging the synergy of Feature Models and Variability Mechanisms, we can model a BPL able to exploit composition, reusability and adaptability typical in product-line engineering for the purpose of increasing process flexibility. Moreover, BPL model is highly maintainable and easy to change for analysts.

However, especially in case of knowledge-intensive business processes, we observed a gap in the changeover from managers (decision-making performers) to technicians (process developers): from the phase of process definition to process implementation, many process characteristics, parameters and decision points can remain tacit and can offer degrees of freedom to developers in choosing the conditions for process configuration and execution. The risk is that at execution time the business process will not comply with the related process model.

We point out that it is essential to elicit all considerable process parameters and decision points coming out during variability modeling and injection and we suggest the adoption of a decision-oriented paradigm. We choose decision tables because they lead to three major advantages:

1. they can be easily updated by decision-making performers;
2. they manage the numerosity of process parameters through a compact and customizable view;
3. they are supported by verification and validation checkers in order to prevent and detect anomalies in decision points.
4. they decrease time consumed to create, modify and update business rules and increase reusability, maintainability and verifiability of them.

3 Decision Tables: Modeling and V&V

By means of decision tables, we can formalize the set of rules which relate the process features to the specific variability mechanisms needed for composition, reuse and adaption of the process assets.

3.1 What They Are

A decision table is a tabular representation of a decision-making task, where the state of a set of conditions determines the execution of a set of actions [15, 16, 17, 18]. In general, a decision table has four quadrants: condition subjects, condition states, action subjects and action values. The table is defined so that, according to the action values, each combination of condition subjects and condition states corresponds to a set of action subjects to carry out.
In the BPL context the decision table adaptation is the following:

— As *condition subjects*: FC = {FC_i} (i=1..n) is the set of the feature categories involved in the BPL;
  - FD = {FD_i} (i=1..n) is the set of features domains,
    with FDi : the domain of FCi (i.e. the set of all possible values);

— As *condition states*: FV = {FV_i} (i=1..n) is the set of feature values sets,
with FV_i = {F_{ik}} (k=1..m_i) : an ordered set of n_i feature values F_{ik}.
Each feature value F_{ik} is a logical expression concerning the elements of FD_i, that determines a subset of FD_i, such that the set of all these subsets constitutes a partition of FD_i ;

— As *action subjects*:
  - VM = {VM_j} (j=1..t), all the possible actions trigging the set of variability mechanisms able to produce the target process;
  - (optional) a set of links to more specific decision table in order to produce more specific sub processes;

— As *action values*: V = {V_j} (j=1..t) relate each feature set to the corresponding action subjects,
with V_j = {true (x), false (-), null (.)}: the set of all possible values of action subject VM_i, which is, in first instance, null for every action subject.

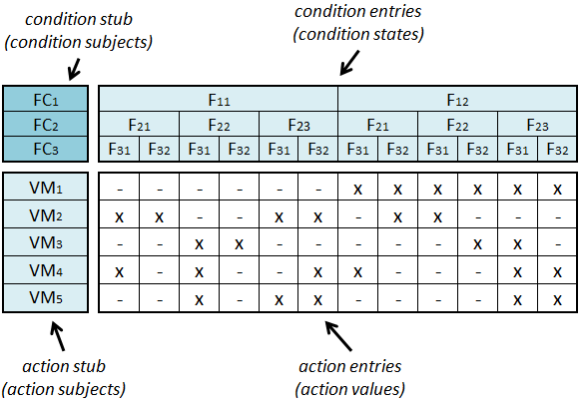


Fig. 7. An example of decision-table in BPL context

A decision table DT can then be defined as a function from the Cartesian product of the condition states FV_i to the Cartesian product of the action values V_j, by which every condition combination is mapped into one (completeness criterion) and only one (exclusivity criterion) action configuration [19] :

DT: $FC_1 \times FC_2 \times ... \times FC_n \rightarrow V_1 \times V_2 \times ... \times V_t$
Each action entry then corresponds to a decision rule.

3.2 What They Do

Using the decision table formalism as a decision modeling tool offers significant advantages in verification, because its structured nature prevents a large number of anomaly types and eliminates the need for a translation into some other operational form, such as Petri nets, first order logic, etc. [20, 21, 22]. This makes it possible to integrate an incremental verification step into the modeling phase itself.

As pointed out in [23] tools that verify rule-bases after operationalizing them into decision table format, generally fail to find anomalies that stretch beyond simple pairs of rules. Therefore, the perspective we adopt here is quite different: we start from the decisions that BPL requires for managing process variability, then we model the decision table and try to ensure it is logically correct from a verification point of view, by operating as much as possible on the table format it was specified in.

It is important that knowledge in specifications is correct, consistent, complete and non-redundant. During and after the building process, the specifications must be verified and validated. This section illustrates how to prevent and detect the decision anomalies and make the BPL decisions non-redundant, complete and consistent.

A) Non redundancy of Decisions. Redundancy usually does not lead to errors in the final system, although it may harm efficiency. The main problem with redundancy, however, is maintenance and the risk of creating inconsistencies when changing the specifications. In BPL some common forms of redundancy:

Subsumption of Rules: in BPL might be rules with conclusions that trig the same variability mechanisms set but with one of them containing additional feature values (and therefore being less general). A more specific case of subsumption is the redundancy of rules (identical pair of column: same variability mechanisms and same features set).

Using decision tables is possible to prevent this kind of anomaly through the checking of *subsumed column pair*, i.e. two table columns specify identical variability mechanisms, while the features set in one column represent a more specific of the application area associated with the other column. Moreover, because in the decision table every possible case is included in only one column (exclusivity), subsumptions and redundancies will not occur.

Irrelevant Features: in a BPL a feature is considered irrelevant if its value does not influence which consequences are being specified: the set of variability mechanisms to be undertaken will be the same regardless of the value of such feature. Therefore, the entire feature category can be considered redundant.

Irrelevant features can easily be found in the *contracted decision table*. This table format, which is obtained by merging neighboring columns with identical action parts (variability mechanisms), will show a "**-**" entry for all feature values associated with an irrelevant feature category. As shown in figure 8 the fourth condition "product type" is irrelevant and then will be eliminated.

C1. auction access	open						closed					
C2. bidding	english		dutch		hybrid		english		dutch		hybrid	
C3. buyer <=> seller	*D*	*I*	*D*	*I*	*D*	*I*	*D*	*I*	*D*	*I*	*D*	*I*
C4. product type	-											
A1. "buyer login" *extends* "login"	-	-	-	-	-	-	x	x	x	x	x	x
A2. "english bidding" *implements* "bidding"	x	x	-	-	-	-	x	x	-	-	-	-
A3. "dutch bidding" *implements* "bidding"	-	-	x	x	-	-	-	-	x	x	-	-
A4. "hybrid bidding" *implements* "bidding"	-	-	-	-	x	x	-	-	-	-	x	x
A5. "hybrid bidding" *inherits* "english bidding"	-	-	-	-	x	x	-	-	-	-	x	x
A6. "hybrid bidding" *inherits* "dutch bidding"	-	-	-	-	x	x	-	-	-	-	x	x
A7. "start buyer" *subs* "start seller"	-	x	-	x	-	x	-	x	-	x	-	x
A8. "start seller" *subs* "start buyer"	-	x	-	x	-	x	-	x	-	x	-	x
A9. "H" jump *subs* "G" jump	-	x	-	x	-	x	-	x	-	x	-	x
A10. "G" jump *subs* "H" jump	-	x	-	x	-	x	-	x	-	x	-	x
A11. "descending" *subs* "ascending" ∈ "EnglishBidding.bid"	-	x	-	-	-	x	-	x	-	-	-	x
A12. "ascending" *subs* "descending" ∈ "DutchBidding.bid"	-	-	-	x	-	x	-	-	-	x	-	x
A13. "price = price + Δ" *subs* "price = price - Δ" ∈ "DutchBidding.markThePrice"	-	-	-	x	-	x	-	-	-	x	-	x
A14. "price > threshold" *subs* "price < threshold" ∈ "DutchBidding.xor2"	-	-	-	x	-	x	-	-	-	x	-	x

Fig. 8. An example of irrelevant feature

B) Completeness of Decisions. Within the specific BPL domain area, the following omissions often occur:

Unused Feature values or Combination: when feature values (or combinations) never occur as premises, a number of rules may be missing. Detecting the completeness of all combinations of feature values could not be simple.

The nature of the decision table easily allows to check for completeness: the number of simple columns should equal the product of the number of states for every condition. This guaranty of completeness of condition combinations is one of the main advantages of decision tables. Furthermore, the *blank columns* into the decision table easy highlight what are the unused combinations of feature values.

Unreachable Variability Mechanisms: when there are mechanisms which are never deduced and then never triggered.

The format of the decision table easily shows unreachable conclusions, *checking for rows with all "-" into the action entries stub*.

C) Consistency of Decisions. Dividing the knowledge over a large number of rules, designed independently, may lead to problems of inconsistency, such as:

Conflict of Rules: in BPL it might be rules with two or more premises containing same features values (or overlapping combinations), but leading to contradictory set of variability mechanisms to be undertaken.

An ambivalent column pair anomaly would refer to a situation in which two table columns specify contradictory actions to be undertaken for overlapping sets of input environments. In a decision table all columns are non-overlapping and each column

refers to exactly one configuration of conclusions, therefore is just necessary to search for potential conflicts only into each single column.

Finally, in order to automate such detection it's possible to use a set of *constraints*. Constraints are introduced into the decision table, in order to express semantic relationships between different decision arguments. These constraints can then be used in order to detect an ambivalent rules, e.g. rules that violate a constraint such as "NOT (VM_i AND VM_j)' (an example in figure 9 and table 1).

C1. auction access	open						closed					
C2. bidding	english		dutch		hybrid		english		dutch		hybrid	
C3. buyer <=> seller	*D*	*I*	*D*	*I*	*D*	*I*	*D*	*I*	*D*	*I*	*D*	*I*
A1. "buyer login" *extends* "login"	-	-	-	-	-	-	x	x	x	x	x	x
A2. "english bidding" *implements* "bidding"	x	x	-	-	-	-	x	x	-	-	-	-
A3. "dutch bidding" *implements* "bidding"	-	-	x	x	-	-	-	-	x	x	-	-
A4. "hybrid bidding" *implements* "bidding"	-	-	-	-	x	x	-	-	-	-	x	x
A5. "hybrid bidding" *inherits* "english bidding"	-	-	-	-	x	x	-	-	-	-	x	x
A6. "hybrid bidding" *inherits* "dutch bidding"	-	-	-	-	x	x	-	-	-	-	x	x
A7. "start buyer" *subs* "start seller"	-	x	-	x	-	x	-	x	-	x	-	x
A8. "start seller" *subs* "start buyer"	-	x	-	x	-	x	-	x	-	x	-	x
A9. "H" jump *subs* "G" jump	-	x	-	x	-	x	-	x	-	x	-	x
A10. "G" jump *subs* "H" jump	-	x	-	x	-	x	-	x	-	x	-	x
A11. "descending" *subs* "ascending" ∈ "EnglishBidding.bid"	-	x	-	-	-	x	-	x	-	-	-	x
A12. "ascending" *subs* "descending" ∈ "DutchBidding.bid"	-	-	-	x	-	x	-	-	-	x	-	x
A13. "price = price + Δ" *subs* "price = price - Δ" ∈ "DutchBidding.markThePrice"	-	-	-	x	-	x	-	-	-	x	-	x
A14. "price > threshold" *subs* "price < threshold" ∈ "DutchBidding.xor2"	-	-	-	x	-	x	-	-	-	x	-	x

Fig. 9. Final decision table for Auction BPL

Table 1. Rules and Constraints

Rules	Constraints
If (C1.2) then A1	A2 xor A3 xor A6
If (C2.1) then A2	A6 only if A4 and A5
If (C2.2) then A3	(A7 only if A8) or (A8 only if A7)
If (C2.3) then A4 and A5 and A6	(A9 only if A10) or (A10 only if A9)
If (C3.2) then A7 and A8	
If (C3.2) then A9 and A10	
If (C2.1 and C3.2) then A11	
If (C2.2 and C3.2) then A12 and A13 and A14	
If (C2.3 and C3.2) then A11 and A12 and A13 and A14	

4 Related Works

Commonality and variability management in a (software or business) product line is a key activity that usually affects the degree to which a product line is successful. First of all, Software Product Line community has spent huge amount of resources on developing various approaches to dealing with variability related challenges over the last decade. From a systematic literature review [24] attempting to identify the kinds of variability models used in the reviewed approaches, it was not a surprise that fourteen approaches used feature modeling according to FODA [12], which firstly

was proposed to address the issues related to commonality and variability in 1990. On the other hand, the decision modeling was used in six approaches, however both approaches are independent of any particular way of modeling variability. So we can state that there is no contribution in literature of mixing feature modeling and decision-oriented paradigms for managing variability and its evolution.

Analyzing world of Business Process Management, many researchers highlight the need for a next generation process management technology, which is by orders of magnitudes more powerful and flexible than contemporary process management systems [25]. There are also some authors that have reported evaluations of existing tools and techniques for managing variability in process models. Aiello et al. [26] reported the evaluation of existing tools and frameworks for variability management. Such evaluation was made considering five requirements stemming from the need of managing evolution of processes with variability. The authors conclude that none of the existing frameworks addresses all or even most of the five requirements. Also La Rosa et al. [27] proposed a method and tool suite for managing business process variability based on configurable process models. Unlike our approach, which is compositional, their approach is annotative, basically wrapping the variability with additional gateways driven by the decision model, cluttering the process model with configuration knowledge details. Other different approaches have been defined to achieve configuration by means of annotations. The PESOA (Process Family Engineering in Service-Oriented Applications) project [13] defines so-called variant-rich process models as process models extended with stereotype annotations to accommodate variability. These stereotypes are applied to both UML Activity Diagrams and BPMN models. Also the authors in [28] propose a representation system called Map to capture variability in process models expressed in an intentional manner through business goals and strategies. However, they don't provide means for separating the analysis and modeling phase from the validation phase where managing process parameters and preventing anomalies in decision points. Therefore, we argue that there is space for extensive research and development in the area of methodology and frameworks for the explicit management of variability - from business processes to software products – by reflecting a product line.

5 Conclusion

The paper addresses the research question, whether and how to capture and model variability in business processes to solve a flexibility problem in changing environments. The authors propose Business Process Line engineering together with decision table-based support. The former takes an inspiration from Software Product Lines, since there is a duality between products and processes, whereas the latter is used to formalize the set of rules that relates the process features to the specific variability mechanisms. Another benefit of the decision tables is that they manage the relationships among business needs, environmental changes and process tasks through a compact and customizable view and are supported by verification and validation checkers that can detect and prevent anomalies in decision points.

In a real case study we practiced the proposed methodology by leveraging the synergy of feature models, variability mechanisms and decision tables. The application of DT-based BPL engineering approach proves that the Business Process Line benefits from fundamental concepts like composition, reusability and adaptability and satisfies the requirements for process definition flexibility. Moreover, decision table formalism enables decision-making performers to implement business processes by themselves and thus decreasing time consumed to create, modify and update business rules and increasing reusability, maintainability and verifiability of them. Our future work will focus on systematic investigation to validate and provide empirical evidence to the benefits of our BPL methodology in industrial case studies, with particular attention to service-oriented enterprises.

As the significant achievements gained by introducing software product lines in the software industry, we expect that narrow and strategic application of these concepts could yield order of magnitude improvements in business process flexibility and a discontinuous jump in competitive business advantage.

References

1. Duan, Y., Huadong, M.: Modelling flexible workflow based on temporal logic. In: The 9th International Conference on Computer Supported Cooperative Work in Design Proceedings, May 24-26, vol. 1, pp. 594–599 (2005)
2. Nurcan, S., Edme, M.H.: Intention Driven Modelling for Flexible Workflow Applications. Special issue of the Software Process: Improvement and Practice Journal on Business Process Management, Development and Support 10, 4 (2005)
3. Hammer, M., Champy, J.: Reengineering the Corporation-A manifesto for Business Revolution. Harper Business (1994)
4. Schnieders, A., Puhlmann, F.: Variability Mechanisms in E-Business Process Families. In: Abramowicz, W., Mayr, H.C. (eds.) Proceedings of the 9th International Conference on Business Information Systems, BIS 2006. Lecture Notes in Informatics, vol. 85, pp. 583–601. GI (2006)
5. Regev, G., Wegmann, A.: A Regulation Based View on Business Process and Supporting System Flexibility. In: Proceedings of the CaiSE 2005 Workshops, vol. 1, pp. 91–98 (2005)
6. Nurcan, S.: A survey on the flexibility requirements related to business processes and modeling artifacts. In: Proceedings of the 41st Annual Hawaii International Conference on System Sciences, Big Island, Hawaii, USA (2008)
7. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. SEI Series in Software Engineering. Addison–Wesley (August 2001)
8. Ommering, R.V.: Building product populations with software components. In: ICSE 2002, Orlando, Florida (2002)
9. Thomphson, J., Heimdalh, M.P.: Extending the Product Family Approach to support n-Dimensional and Hierarchical Product Lines. In: 5th IEEE Symposium on Requirements Engineering, Toronto, Canada (2001)
10. Svahnberg, M., Van Gurp, J., Bosch, J.: On the notion of variability in Software Product Lines. In: Working IEEE/IFIP Conference on Software Architecture (2001)
11. Pohl, K., Böckle, G., Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer Verlag New York, Inc. (2005)

12. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study, Technical Report CMU/SEI-90-TR-021, SEI, Carnegie Mellon University (November 1990)
13. Schnieders, A., Puhlmann, F.: Variability Mechanisms in E-Business Process Families. In: Proceedings of the 9th International Conference on Business Information Systems, BIS 2006, Klagenfurt, Austria, pp. 583–601 (2006)
14. Malone, T.W., Crowston, K., Herman, G.A.: Organizing Business Knowledge The MIT Process Handbook. MIT Press, Cambridge (2003)
15. Vanthienen, J., Mues, C., Wets, G., Delaere, K.: A tool-supported approach to inter-tabular verification. Expert Systems with Applications 15, 277–285 (1998)
16. Maes, R., Van Dijk, J.E.M.: On the Role of Ambiguity and Incompleteness in the Design of Decision Tables and RuleBased Systems: The Computer Journal 31(6) (1988)
17. Ho, T.B., Cheung, D., Liu, H.: Advances in Knowledge Discovery and Data Mining. In: 9th Pacific-Asia Conference, Vietnam (2005)
18. Bar-Or, A., Keren, D., Schuster, A., Wolff, R.: Hierarchical Decision Tree Induction in Distributed Genomic Databases. IEEE Transactions on Knowledge and Data Engineering 17 (2005)
19. Antoniu, G., van Harmelen, F., Plant, R., Vanthienen, J.: Verification and Validation of Knowledge-Based Systems: Report on Two 1997 Events. AI Magazine 19(3), 123–126 (1998)
20. Larsen, H., Nonfjall, H.: Modeling in the Design of a KBS Validation System. Int. Journal of Intelligent Systems 6, 759–775 (1991)
21. Zhang, D., Nguyen, D.: A Tool for Knowledge Base Verification. IEEE Transactions on Knowledge and Data Engineering 6(6), 983–989 (1994)
22. Zlatareva, N.: A Framework for Verification, Validation, and Refinement of Knowledge Bases: The VVR System. Int. Journal of Intelligent Systems 9, 703–737 (1994)
23. Preece, A., Shinghal, R.: Foundation and Application of Knowledge Base Verification. Int. Journal of Intelligent Systems 9, 683–701 (1994)
24. Chen, L., Ali Babar, M., Ali, N.: Variability management in software product lines: A systematic review. In: SPLC 2009, San Francisco, CA, USA, pp. 81–90 (2009)
25. Dadam, P., Reichert, M.: The ADEPT project: a decade of research and development for robust and flexible process support. Computer Science - R&D 23(2), 81–97 (2009)
26. Aiello, M., Bulanov, P., Groefsema, H.: Requirements and Tools for Variability Management. In: 34th Annual IEEE Computer Software and Applications Conference Workshops, COMPSACW, pp. 245–250 (2010)
27. Rosa, M.L., Dumas, M., Hofstede, A., Mendling, J.: Configurable multi-perspective business process models. Information Systems 26(2) (2011)
28. Rolland, C., Nurcan, S.: Business Process Lines to Deal with the Variability. In: Proceedings of 43rd Hawaii International Conference on System Sciences, HICSS (2010)

Author Index