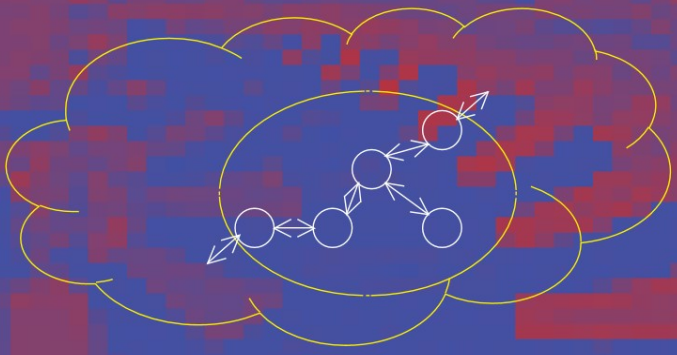
State-of-the-Art Survey

LNCS 5380

Martin Wirsing
Jean-Pierre Banâtre
Matthias Hölzl
Axel Rauschmayer (Eds.)

Software-Intensive Systems and New Computing Paradigms

Challenges and Visions



Springer

Lecture Notes in Computer Science 5380

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

Martin Wirsing Jean-Pierre Banâtre
Matthias Hölzl Axel Rauschmayer (Eds.)

Software-Intensive Systems and New Computing Paradigms

Challenges and Visions

Springer

Volume Editors

Martin Wirsing
Institute of Computer Science, LMU Munich
Munich, Germany
E-mail: wirsing@pst.ifi.lmu.de

Jean-Pierre Banâtre
University of Rennes I and INRIA / IRISA
Rennes Cedex, France
E-mail: jpbanatre@inria.fr

Matthias Hölzl
Institute of Computer Science, LMU Munich
Munich, Germany
E-mail: matthias.hoelzl@ifi.lmu.de

Axel Rauschmayer
Institute of Computer Science, LMU Munich
Munich, Germany
E-mail: axel.rauschmayer@ifi.lmu.de

Library of Congress Control Number: 2008939131

CR Subject Classification (1998): D.2, D.1.5, D.3, F.3.1-2

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-89436-5 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-89436-0 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12566389 06/3180 5 4 3 2 1 0

Preface

Software-intensive systems have become increasingly important for a multitude of products and services from all sectors of the economy, our national and international infrastructure, and our daily lives. The ongoing decrease in size and cost of microprocessors and storage devices is leading to the development of ever more distributed and decentralized systems. Systems are assembled as dynamic federations of autonomous and evolving components instead of monolithic applications, they perform tasks of staggering complexity with continuously changing requirements and in a permanently evolving environment. In the near future novel technologies will allow the construction of systems with millions of nodes, and systems will be likely to contain subsystems based on new computing paradigms such as molecular computing.

To identify these emergent trends, their impact on the information society in the next 10–15 years, and the challenges they present to computing, software engineering, cognition and intelligence, the European Commission has established two Coordinated Actions: initially the project "Beyond the Horizon"[1] and then, starting in 2006, the project "InterLink"[2]. Both projects are coordinated by the European Research Consortium for Informatics and Mathematics (ERCIM EEIG) and funded by the Future and Emerging Technologies (FET) Unit of the European Commission. The ongoing project InterLink is composed of three thematic working groups: software-intensive systems and new computing paradigms; ambient computing and communication environments; intelligent and cognitive systems.

This volume presents the results of the working group on software-intensive systems and novel computing paradigms. The objective was to imagine the landscape in which the next generations of software-intensive systems will operate. To this end three workshops were organized on this topic. Participation in the workshops was by invitation only. Over 30 leading researchers from Europe, Asia, Australia, USA, and Canada presented and discussed future R&D directions, challenges, and visions in the emerging areas of software-intensive systems and new computing paradigms. Each workshop was structured by a three-step process: At first the participants presented those topics and developments they considered to be the most interesting and challenging in the field. Then the participants split into working groups according to the central themes that had been identified in the initial presentations. In a concluding plenary session the results of the working groups were integrated.

From the beginning of the workshops it was evident that future software-intensive systems will feature massive numbers of nodes per system, operate

[1] http://beyond-the-horizon.ics.forth.gr/
[2] http://interlink.ics.forth.gr/

in open, non-deterministic environments, deal with large amounts of data, interact with humans or other software-intensive systems, and adapt to new requirements, technologies or environments without redeployment. To characterize software-intensive systems with these properties, the workshop participants agreed on the term "ensembles." Key research topics comprise the design of emergent systems, management of uncertainty, dependability and trustworthiness of ensembles.

Another important aspect is the development of self-organizing systems with autonomic behavior. Present programming paradigms are less and less adequate the more autonomous software-intensive systems become. New paradigms have to be invented, implemented and evaluated in order to develop high-quality and efficient "ensemble computing systems." Unconventional computing paradigms inspired by biology, chemistry, life or nature are active areas of research areas. The field is now mature enough that true applications in realistic environments can be built and also deployed on traditional Von-Neumann architectures.

This volume starts with an overview of the current state of the art and the research challenges in engineering software-intensive systems. The remainder of the book consists of invited papers of the working group participants and is structured in three major parts: ensemble engineering, theory and formal methods, and novel computing paradigms. These papers cover a broad spectrum of relevant topics ranging from methods, languages and tools for ensemble engineering, socio-technical and cyber-physical systems, ensembles in urban environments, formal methods and mathematical foundations for ensembles, orchestration languages to disruptive paradigms such as molecular and chemical computing.

Many persons contributed to the success of our workshop series. We offer sincere thanks to all of them. We are particularly grateful to Jessica Michel, Patricia Ho-Hune, and Florence Pesce of ERCIM for their invaluable work and effort in preparing and organizing the workshops. Their friendly manner and managerial skills contributed a great deal to the success of the workshops. We thank our workshop hosts in Urbana-Champaign, Grigore Rosu and José Meseguer, for a productive and friendly work environment. The InterLink project would not have been possible without the scientific coordination of Constanine Stephanidis, Dimitris Plexouxakis, and Antonis Argyros. We thank the EC project officers Wide Hogenhout, Thomas Skordas, and Walter van der Velde for their continuing encouragement and support. We are also grateful to Springer for their helpful collaboration and assistance in producing this volume. Our sincere thanks go to all authors for the high quality of their scientific contributions and for accommodating our tight schedule. Finally, we thank all workshop participants for the lively discussions and their deep insights into the subject matter.

September 2008

Martin Wirsing
Jean-Pierre Banâtre
Matthias Hölzl
Axel Rauschmayer

Coordination Action InterLink

The Coordination Action InterLink "International Cooperation Activities in Future and Emerging ICTs" (Contract No. 034051, 2006-10 – 2009-03) is funded by the Future and Emerging Technologies (FET) Programme of the European Commission.

Scientific Coordination: Institute of Computer Science, Foundation for Research and Technology, Greece (ICS-FORTH) Contact: Constantine Stephanidis.

Administrative and Financial Coordination: ERCIM EEIG. Contact: Jessica Michel.

InterLink Working Groups

1. Software-intensive systems and new computing paradigms
   Working Group leader: Martin Wirsing (LMU Munich, Germany)
   Deputy leaders: Jean-Pierre Banâtre (Université de Rennes 1 and INRIA, France), Matthias Hölzl (LMU Munich, Germany)
2. Ambient computing and communication environments
   Working Group leader: Norbert Streitz (Fraunhofer IPSI, Germany)
   Deputy leader: Reiner Wichert (Fraunhofer IGD, Germany)
3. Intelligent and cognitive systems
   Working Group leader: Rüdiger Dillmann (University of Karlsruhe, Germany)
   Deputy leader: Tamim Asfour (University of Karlsruhe, Germany)

Members of Working Group 1

- Gul Agha
- Jean-Pierre Banâtre
- Gabriel Ciobanu
- José Fiadeiro
- Pascal Fradet
- Jean-Louis Giavitto
- Fausto Giunchiglia
- Seth Goldstein
- Manuel Hermenegildo
- Teruo Higashino
- Matthias Hölzl
- Stephan Jähnichen
- Michael Johnson
- Insup Lee
- Zhiming Liu
- Vicenzo Manca
- José Meseguer
- Jayadev Misra
- Oscar Nierstrasz
- Axel Rauschmayer
- Mike Reed
- Wolfgang Reif
- Jeff W. Sanders
- Heinrich Schmidt
- Lui Sha
- Doug Smith
- Graeme Smith
- Darko Stefanovic
- Carolyn Talcott
- Christof Teuscher
- Martin Wirsing

Table of Contents

III Novel Computing Paradigms

Engineering of Software-Intensive Systems: State of the Art and Research Challenges

Matthias Hölzl, Axel Rauschmayer, and Martin Wirsing

Institut für Informatik, Ludwig-Maximilians-Universität München

Abstract. Software-intensive systems become more and more important in our everyday lives. But their increasing complexity makes it difficult to develop and maintain them. This chapter gives an overview of the state of the art of building software-intensive systems and outlines research challenges that have been identified by the InterLink working group "software-intensive systems and new computing paradigms".

1 Introduction

Developments in ICT have a large influence on the gains in productivity and prosperity that society has seen in recent years [1]. This development is responsible for software being in control of increasing numbers of systems whose failure may have critical consequences—for infrastructure, economy, or the safety of human lives.

These systems are becoming increasingly distributed and decentralized, assembled as dynamically changing orchestrations of autonomous services, expected to adapt to continuously changing requirements and environments. Recent developments in the design of microprocessors and in emerging technologies, such as nanotechnology, will lead to massive increases in the number of nodes of a typical system, while potentially increasing the expected failure rate of individual components. Subsystems based on new paradigms, such as molecular computing or quantum computing, may transition from research laboratories to commercial products in the next years.

It is imperative that we develop the engineering techniques to reliably design, develop and deploy these novel systems. In this chapter, we summarize the current state of the art in the area of engineering software-intensive systems and present research challenges.

Scope

The area of "software-intensive systems" encompasses many large and active fields of industrial practice and academic research, ranging from systems and software engineering to network technologies and hardware. It is also closely connected to fields such as control theory and cybernetics; some research directions for software-intensive systems are directly or indirectly inspired by chemistry, biology, the social sciences and many other areas. Given the enormous scope of

M. Wirsing et al. (Eds.): Software-Intensive Systems, LNCS 5380, pp. 1–44, 2008.

the field it is obviously impossible to detail the state of the art of the whole field, or even to produce an exhaustive survey of all concerned areas. On the other hand, just focusing on a few areas that we consider important or interesting would not serve the purpose this chapter. We have therefore tried to reach a compromise by surveying all topics that are directly relevant to the engineering of software-intensive systems and giving enough pointers to the literature that the interested reader can explore the topics in more depth.

Structure of This Chapter

This chapter consists of three main sections; the first two are concerned with the state of the art of software-intensive systems, the third outlines research challenges: In Sect. 2, "Foundations", we present properties of systems and areas of research which are relevant to the engineering of software-intensive systems. In Sect. 3, "Engineering", we focus on the engineering process itself, in particular on software engineering processes, tools and techniques. The distinction between these two parts is not precise and does not represent a judgement on the importance of the individual areas. For example, security might as well have been in the engineering part as security aspects pervade many areas of systems engineering, and clearly it is one of the most important properties for many systems. Sect. 4 presents several research challenges that we expect to have great impact on the development of software-intensive systems in the next 10–15 years.

The "state of the art" presented in this chapter can roughly be divided into three categories:

- *State of the practice:* the processes, techniques and tools used to develop software-intensive systems.
- *Incremental research:* research about software-intensive systems in areas which are believed to be well-understood, where the direction of the research is relatively clear and most research contributions are of an incremental nature.
- *Research frontier:* research areas where we may have an idea of the goal we are trying to reach but where we do not yet know how to reach that goal and whether the desired result is achievable.

We address all three categories, without explicitly indicating for every topic in which category we consider it to belong. In general, most of our effort is focused on the research frontier.

Related Work

One of the most important European publications in the state-of-the-art assessment of software-intensive systems is without doubt the ITEA Technology Roadmap for Software-Intensive Systems [2]. This report structures the field of software-intensive systems into four *software-technology clusters*, each with several *technology categories* and presents an overview of the state of the art, and short term, mid term and long term challenges for technologies in these categories.

This chapter was strongly influenced the work of WG6 of the IST-FET Coordinated Action "Beyond-the-Horizon." The final report [3] identifies several important research directions and topics in the field of software-intensive systems.

One resource that provided important material for the the state of the practice was the IEEE Software special issue about this topic [4] which contains relevant articles about variations in software development practices, the different software development approaches adopted in different countries [5], the use of documentation [6], requirements engineering [7], software reviews [8], product-line engineering [9], embedded software engineering [10], "Internet speed" software development [11], and the relationship between state of the art and state of the practice [12]. These reports are particularly valuable since most published literature is concerned with "best practice" and not the "actual practice."

2 Foundations

The following topics cover important foundations for the engineering of software-intensive systems which have a direct influence on the engineering process. Progress in research or practice in one of these areas impacts the engineering of systems. We have structured Part 2 as follows:

- *Devices:* the individual nodes present in a software-intensive system.
- *Content:* data, information and knowledge which is acquired, stored and processed by the system.
- *Interoperability and Interaction* between systems as well as between humans and systems.
- *Adaptation* to new environments, new requirements or new regulations.
- *Assurance:* Quality of service and experience, security, dependability.

These areas are not a complete partition of the foundations for software-intensive systems: on the one hand there is a certain amount of overlap between the individual areas, on the other hand there are many links between the areas and it is not always clear into which area an individual research challenge falls. Therefore the division should be understood as a presentational device, not as a proposal for a taxonomy of the field. Our division is slightly different from the one adopted in the ITEA report [2], reflecting the facts that we are more focused on long-term research and general software-intensive systems, whereas ITEA has a stronger emphasis on short or mid-term research and embedded systems.

Furthermore we are concerned almost exclusively with technical matters which impact the development of software for systems; we do not address other topics that clearly play an important role in the engineering of systems, such as legal issues, the psycho-social context, or hardware design.

2.1 Devices

The development of software-intensive systems is in large parts driven by improvements in the available hardware for programmable controllers and computers: the dramatic decrease in cost, size, and energy consumption of integrated

circuits and the availability of computational elements based on new principles enables engineers to control devices with software for which this was not economically feasible only a few years ago; the widespread integration of software in devices in turn enables novel features for many classes of devices, such as network connectivity for household appliances.

The use of software as control mechanism has many advantages for the development of systems, as software can easily implement complex control behaviors that would be difficult to realize in hardware. Furthermore, software is more malleable than hardware and can therefore better serve as the "glue" between individual components. On the other hand, the increasing use of software in systems poses some new problems: the more complex behaviors of individual components may lead to more difficulties in determining and controlling the overall system behavior and to unintended interactions between components.

In the following sections we examine the state of the art of traditional microcontrollers and CPUs, of micro- and nano-scale devices, and of synthetic biology. We also give a very short overview of devices based on novel computing paradigms and sensors.

Miniaturized Devices. In the last 40 years the number of transistors that can be (inexpensively) placed on an integrated circuit has been growing exponentially [13], and it is expected that this trend will continue for some more time to come [14]. This has led to very inexpensive microprocessors and micro-controllers that enable designers to use flexible software-based control for devices which were previously purely mechanical and enabled many new devices, such as cellular phones or PDAs. In addition to having more devices with computer control, computers in embedded devices are becoming increasingly more powerful [15]. A state-of-the-art assessment for integrated circuits is beyond the scope of this chapter but state-of-the-art reports for various application domains are readily available, e.g., [16]. The increasing integration of computational elements with physical artifacts is sometimes called cyber-physical systems [17], an overview of the state-of-the art and planned research is [18].

The increasing number of transistors on integrated circuit also means that support for multiple parallel threads or processes is becoming more prevalent on desktop computers [19], techniques such as hyper-threading or multiple cores are available on most current microprocessors for desktops and servers.

The trend towards more integration is taken even further in the area of miniaturized devices with the integration of whole systems or network into a single device, called Systems-on-Chip (SoC) or Networks-on-Chip (NoC) [20]. Micro-fabrication technology is sufficiently advanced that systems-on-chip can be built which consist of electronics, sensors and actuators on a single silicon substrate. In this case they are commonly called Micro-Electro-Mechanical Systems (MEMS) [21,22,23,24,25]. MEMS are currently used in products as varied as accelerometers, gyroscopes, or the print-heads of ink-jet printers, to name only a few areas. Some of the research areas currently pursued in the MEMS sector are improvements in micro-fabrication technology, both for high- and low-volume production of MEMS, as well as packaging of MEMS.

The large number of chips deployed in systems means that the issue of power consumption becomes more important. Devices which consume less energy are not only cheaper and more environmentally friendly they also provide advantages from a purely technical point of view, as they generate less heat and require less battery power. Many power reduction techniques for microprocessors are currently employed and developed, [26] presents an overview of the state of the art in the area. The development of power sources for mobile and wireless systems is a task which is actively pursued both industry and research, partly by improvements to the power density of batteries and partly by the development of novel power sources such as fuel cells.

Further miniaturization of systems can be expected from the emerging field of nanotechnology and the corresponding area of Nano-Electro-Mechanical Systems (NEMS). In contrast to microelectronics there are, as yet, few commercial applications of nanotechnology outside of nanomaterials. Nanoelectronics can be used to build miniaturized replacements for components such as FPGAS out of nano-material [27], but also to improve certain aspects of more traditional design, e.g., interconnects in integrated circuits [28]. As with microelectronics, a state-of-the-art assessment of nanotechnology is beyond the scope of this chapter.

There are several implications that increasing miniaturization has for the area of system design: the most obvious and visible aspect is that systems now generally contain several or even large numbers of controllers and programmable components, with widely differing computational power, see Section 3.2. Many systems now include sensors as essential components, see Section 2.2.

Another important trend is the increasing adaptation of systems and components to their particular task and the flexible assembly of parts with limited functionality into more powerful ensembles. Examples are reconfigurable computing [29], micro-robotics, and claytronics [30,31,32].

The growing number of system components and the inherent defect rate in the production of micro- and nano-scale systems mean that future systems will have to deal with a higher percentage of inoperative nodes, and more frequent failures of individual nodes in the course of their operation. Inoperative nodes in nano-devices can be identified using tests and then bypassed by post-fabrication configuration. To be feasible this will require changed design-flows for these devices [33]. But miniaturization also leads to increased sensitivity to the environment and a higher rate of transient errors. Therefore miniaturization requires additional design efforts to design dependable systems [34]. Current research in the area concerns, e.g., the design of fault-tolerant virtual reconfigurable circuits [35], self-replicating hardware [36], or self-organizing SIMD architecture [37].

Devices Based on Novel Computing Paradigms. There is ongoing research in the area of novel computing paradigms. We will address the software-development issues and opportunities of these approaches in Part 3, in this section we are concerned with devices built out of non-traditional material, e.g. DNA. In general, these areas are not yet far enough developed to have industrial applications, but interesting results have been obtained in research labs.

Research for building and manipulating objects at the nanoscale level in physics, chemistry and molecular biology reveals promising approaches to unconventional computing [38, 39], as foreseen by R. Feynman [40] and later popularised by E. Drexler [41].

Microfluidics-based biochips provide new kinds of sensors for biochemical analysis; biomechatronics aims to integrate bio-sensors and -computers into the human body for rehabilitation and augmentation purposes. In both areas rapid progress is made in research and experimental trials [42, 43], and commercial applications can be expected in the next few years. It is, at this point of time, not entirely clear how these developments will influence the design of software-intensive systems, but in particular military applications of biomechatronics seem likely.

Another important area of molecular computing is concerned with using DNA for computational purposes [44]. While there are efficient DNA computers for certain specialized problems, molecular computing is still subject to the same theoretical limitations as classical computation, although it is possible to build computing devices with massive parallelism.

In the field of molecular biology, harnessing molecules to compute can be traced back at least to [45] and research in this field explodes with the landmark experiment of L. Adleman [46]. Molecules can be used for their physical interactions or their chemical reactions [47, 48, 49] or, in a biological context, using the gene regulation machinery of a cell to achieve some computations [50, 51, 52]. These computations can be designed and implemented directly "by hand" or using directed evolution [53, 54].

In this context, *synthetic biology* [55, 56, 57] emerges as a new engineering discipline at the convergence of genetic engineering and computer science. It encompasses the design and the implementation of complex artificial biological systems for a variety of applications. The web site [58] is a good introduction. The web site of iGEM, the international Genetically Engineered Machine competition, [59] gives an outline of the envisioned applications. In this new area, the pace of the technological changes seems to be consistent with Moore's law [60]. Computer science is relevant for their development at two levels. At the engineering level it addresses the topics of:

- how to specify and design a set of standard (biological) components with well-defined characteristics and performances that can be used and reused in the building of (artificial) biological systems;
- how to hierarchize and compose these biological components;
- which design methodology and computer tools can help the development of these kind of systems;
- how to reverse-engineer existing biological modules to optimize them and to adapt them to the needs of synthetic biology.

At the programming level, synthetic biology poses many of the challenges sketched in the previous section: a massive number (billions) of unreliable elementary entities that interact and cooperate dynamically and randomly. It is

nevertheless necessary to ensure a global emergent behavior, robust and persistent in time, that can serve as a foundation for computing.

The main question is how to "instruct" the enormous population of elementary entities to obtain a global and coherent behavior from the local regulation (interaction, cooperation, development) of the elementary processes. It is crucial to note that this is similar to the problems encountered in large software-intensive systems. Recognizing this fact, emerging new research fields such as amorphous computing, spatial computing, autonomic computing, organic computing, etc., are investigating properties of artificial systems which are usually attributed to living systems such as self-organization, self-healing, self-optimization, or sustainability.

Another important non-traditional computation paradigm is quantum computing, where quantum-mechanical phenomena are used as essential ingredients of computations. These computers may offer large theoretical complexity improvements on certain classes of problems, such as the factorization of large integers and computing discrete logarithms, and they offer provable complexity advantages for at least one other problem, quantum database search. The former would obviously influence the design of software-intensive systems if quantum computers became available commercially, since public-key methods are often based on one of these mechanisms [61]. For a more detailed description of the capabilities of quantum computers and their applicability to different areas we refer to the IST-FET publication [62], the ERA-Pilot Project [63], and [64].

2.2 Content

In addition to inexpensive and miniaturized devices, content is one of the main drivers for the development and adaptation of new software-intensive systems. The main trend is to go from dealing with raw signals or data to information and knowledge. We adopt the ITEA definitions of data as "raw representation in binary format," information as "representation of knowledge that can be understood by a user," i.e., data that is structured, annotated with metadata and that possesses a well-defined semantics, and knowledge as "information that is of interest to users." In general, according to this point of view, information and knowledge can only be distinguished by taking into account the point of view of the user of the data, not by their representation. See [65] for a more in-depth treatment of this question. While we are aware that these definitions are in many ways problematic and the assignment of data into the various categories to a certain extend arbitrary, the concepts of data/information/knowledge nevertheless define an important conceptual distinction and it seems beneficial to use them in a way that is consistent with other existing documents.

We have structured this section, similar to the corresponding technology clusters in the ITEA roadmap, into sections about content acquisition and processing, content representation, and data and content management.

Content Acquisition and Processing. The acquisition of content can be divided into two distinct areas which nevertheless share many of the same issues:

acquiring data from sensors and acquiring data that already exists in digital form in the system's environment.

The improvements in the production of MEMS allow the wide deployment of sensors and sensor networks. Sensor networks are widely used in areas such as environmental monitoring, 3D shape recognition or target detection and tracking [66,67,68,69,70]. Fusion of sensor data from diverse sensors and sensor networks can be used to achieve functionality that surpasses that possible from single sensors. Mechanisms to integrate data from different sensors are already widely used in industrial and military applications, but data fusion is also a currently very active field of research [71,72]

In many wireless sensor networks it is not easily possible to replace the batteries of the sensors. Therefore energy-efficiency is an important issue. Efficient use of the limited energy budget can be made by optimizing sensor placement [73], possibly taking into account irregularities in the environmental conditions [74], or by improved power management [75,76]. Reconfiguration of the sensor network is another method to reduce power consumption which can additionally be used to increase performance [77], or provide a level of self-organization [78] to the network.

Closely related to the issue of power consumption are techniques for ensuring coverage of the sensor network's operational area [79,80]. Most often designers of wireless sensor networks try to reduce superfluous redundancy as this negatively affects the energy budget and simultaneously increases network congestion [81].

When integrating the data of many sensors operating independently, it becomes important to ensure that the individual data items are temporally [82, 83,84] and spatially [85] correlated, and to monitor the correctness of the transmitted data [86]. Many sensor networks are deployed in hostile environments. Here, in addition to ensuring the correctness of the data, self-protection of the network becomes another important issue [87].

Another important aspect for sensor data is the context-sensitivity of capturing and interpretation. It is in general not practical or desirable to permanently capture the data from all available sensors. Therefore software-intensive systems have to be able to derive their current and future operating context and decide which data should be retained, which data should be processed and summarized, and which data should not be captured.

Given the large amount of data available both from sensors and other sources it becomes more important to automatically "understand" the data, i.e., to analyze trends, to make predictions, or to derive abstractions from data and information without or with little human interventions. There is significant scientific progress reported in this area [88,89,90,91,92] but the application of these techniques to currently deployed systems seems to be limited to specialized areas. However, because of the large possible impact it is likely that the research will be adopted by more commercial products in the next years.

Content Representation. Over the last decade, many successful standards for representing structured and semi-structured data and metadata have been defined and widely adopted in industrial applications. The most significant is

the XML standard [93] and a series of related standards, such as namespaces, inclusions (XInclude), the XML Information Set (Infoset), etc. See [94] for a complete list of the relevant standards.

Similar standards for better data modeling such as the Meta Object Facility (MOF, [95]) and *Resource Description Framework* (RDF) [96] have been defined and adopted by the industry, but they are not as popular as XML. An ongoing area of research are *ontology languages* for formally describing knowledge. Examples include the Knowledge Interchange Format (KIF, [97]), the Cyc Representation Language (CycL, [98]), Description Logics (DL, [99]) and the DL-based Web Ontology Language (OWL, [100,101]). No clear winner among these languages seems to be emerging, but there are efforts underway to unify them, e. g. N3Logic [102], Simple Common Logic [103], and the Ontology Definition Metamodel (ODM, [104]).
Research challenges include:

- Constructing knowledge bases: How can we codify information so that it can be used to solve problems automatically? Existing (and continually updated) examples include Cyc [105,106], Wordnet [107], and the Suggested Upper Merged Ontology (SUMO, [108]).
- Managing ontologies: Ontology debugging [109], ontology visualization [110], ontology evolution [111], and ontology versioning [112].
- Mapping natural language texts to and from ontologies. Less ambitious, partially solved versions of this problem are:
  - RDFa allows one to reversibly merge text and data for publication on the web [113].
  - *Controlled natural language* can be used to specify and query knowledge [114,115].
  - With the recent popularity of tagging, some semantic information is already available (in weblogs, photo collections, etc.) [116].
- Beyond ontological reasoning: Many application domains necessitate knowledge representations and reasoning that go beyond what standard ontologies provide. Examples include geospatial data and fuzzy time data [117].
- Enhancing applications such as integrated development environments or even office suites with semantic technology.

Other visible trends are the integration of multimedia content and active content into systems, and the separation of content from presentation. These trends are, of course, inspired by the availability of network bandwidth and the increasing variety of devices on which content is consumed. In both areas mature standards exist, e.g., GIF, PNG, JPEG, MP3, OGG or MPG for images, audio and video content, ECMAScript-enabled browsers with XHTML and CSS for active content with separated presentation. Proprietary solutions such as Java (e. g. via BluRay), Adobe Flash or Microsoft Silverlight will also play an important role in these areas.

Data and Content Management. The management of data and content is an area where the increasing distribution of systems poses great challenges

with issues ranging from the identity of data, systems and people over digital rights/restrictions management and privacy to search and resource management.

Even centralized systems have issues with the identity of data items: databases often contain multiple entries for single items. This problem is compounded in federated information systems (FIS), where a single data item may be divided between different nodes of the system [118]. Furthermore, the integration of heterogeneous data with independently evolving schema representations is still an active area of research with no universally applicable solutions.

Consistency and integrity of data are more difficult to ensure in distributed systems, in particular in systems operating in open environments in which nodes may dynamically join and leave the network and where individual nodes may have competing goals. This topic is addressed further in Section 2.4.

The increasing variety of devices on which multimedia content can be displayed and the wide range of their capabilities gives rise to the recoding of content, either to upgrade the content quality of stored data without changing its meaning, or to reduce the transmitted data for devices with limited capabilities or slow network connections. Various commercial and open source solutions exist for the latter application, but upgrading content quality remains a mostly manual process. Related to this issue is the certification of content: currently there exist reliable mechanisms for public exchange of keys and for signing files, but no adequate technologies to certify the content of a file independent of its encoding in a particular format.

Another content management issue that has been investigated for a long time is the protection of content and the management of digital rights and restrictions [119]. While both commercial and academic research in this area have increased the capabilities of content protection technologies there are virtually no examples of software or content that has not been duplicated without the copyright holder's permission shortly after it was made public. The currently most successful approaches rely on either permanent or periodic connections to the Internet, or a hardware "dongle" which either contains data that is important for the execution of the application, or which executes important parts of the application logic. This situation is even more pronounced for content which is intended for consumption by people, such as music or videos. Here most approaches to enforce copyrights have been unsuccessful while imposing significant inconveniences on legitimate users, and currently the trend in the music business seems to be to provide unprotected content on disks and even online. A related issue is watermarking [120] of digital content which enables the embedding of information of metadata into media files such that it cannot be easily removed.

The possibility to store and process previously unimaginable amounts of data also poses new challenges in the related areas of data mining, search, and privacy. On the one hand, the large amounts of cheaply available data and processing capability and the integration of previously unconnected databases enable many new forms of data mining technology, and many businesses routinely mine their databases for purposes ranging from customer satisfaction analysis to fraud recognition. However, this amount of data mining also raises serious privacy

concerns which are currently only partially addressed by technical and regulatory measures [121].

Other issues arising from the increasingly large amounts of distributed data in modern software-intensive systems are resource management, including data backups, and retrieving data. Resource management concerns the traditional topics pertaining to data management in federated information systems, such as the distribution of data to the individual nodes, horizontal and vertical partitioning of data, but increasingly also issues such as retention of data according to various regulations and laws and the related topic of "distributed garbage collection," i.e., identifying data that is no longer needed and can be deleted. Pragmatic solutions for these problems exist, but they are often very labor-intensive and not amenable to automation.

Currently data retrieval from databases or semi-structured data collections is either performed by queries in languages such as SQL or by syntactic keyword searches. Many research efforts to improve the quality of searches exist, e.g., in the areas of adaptive information extraction [122], association mining [123], software retrieval services [124], or indexing for search [125], graph mining [126], and in related areas such as autonomous hypertext authoring [127], evaluation of search services [128], interestingness measures for data mining [129]. That research is currently not widely used in industrial applications [116]. Similar observations can be made for the integration of different data sources [130].

2.3 Interoperability and Interaction

An important aspect of software-intensive systems is the interoperation of distributed components and the interaction of users with the system. In the following sections we present the aspects of networking, system-system interaction and human-system interaction.

Networking. Today's network infrastructure seems to increasingly develop into the direction of "IP everywhere" with a host of different underlying infrastructures such as Ethernet and wireless LAN. The identification of devices can therefore often be performed by their IP address. The well-known shortage of IPV4 addresses has been addressed by the development of the IPV6 protocol, which is deployed on current network equipment and personal computers, and also increasingly on networked embedded devices. However adaption of IPV6 has been slow, most networks still rely on IPV4. An overview of the current issues in the development of networks can be found in [131].

One trend is the increasing importance of mobile and ad-hoc networks and their particular properties [132], e.g., scalability of wireless networks [133], inference analysis [134], or delay and capacity trade-offs [135]. A current research area which is particularly relevant for wireless networks is the topology of networks, e.g., controlling the network topology [136], or topology aggregation techniques [137]. Another topic is quality of service (QoS), which we address in Sect. 2.5.

The increasing number of personal devices and computers with network access has led to the growing importance of networks which are not structured according

to the client/server paradigm. Peer-to-peer networks and data grids are the most prominent examples, a survey of peer-to-peer content distribution networks is [138], a survey of data grids is [139] . Since they no longer rely on central servers such networks present new challenges, for example some nodes in peer-to-peer networks may deliberately introduce mislabeled content, techniques to avoid this are, e.g., ranking of peers [140]. Many nodes in current networks can perform multiple functions, as client, server, router, etc. The looser definition of the roles necessitates a more dynamic configuration of the network, e.g., by automated resource announcement and discovery, by auto-configuring and self-managing nodes.

An interesting aspect of software-intensive systems is the combination of network transparency and location awareness. Location awareness exists to a limited extend, e.g., by automated discovery of Bluetooth devices or printers. However, current systems generally have too little knowledge about the state and desires of their users to consistently take the right decisions about device selection; improvements in location awareness will therefore be closely related to improvements in human-computer interaction and semantics-based interactions, see Section 2.3.

The wide range of different devices and network connections varying several orders of magnitude in throughput and latency necessitate differences in the communication styles and in the data sent over the network. Currently this scaling is mostly done manually, either on the server side or even by clients having to select different resources according to their capabilities.

System-System Interaction. Interactions between nodes of software-intensive systems and between different systems are today commonly based on cross-platform integration technologies, such as web services or CORBA, with standards such as XML [93] serving as data interchange format. However, integration of heterogeneous data from different systems generally has to be performed manually, e.g., by writing XSLT transformations between the different data representation. Checks for correctness and consistency of conversions are usually purely syntactic.

Coordination of different nodes or interacting systems is often programmed in coordination or orchestration languages such as WS-BPEL [141], which are essentially programming languages augmented with some features for programming compositions and possibly integrated into a middleware. Orc [142,143] is an orchestration language that defines a minimal set of primitives for orchestrations with a well-defined semantics. Service-level and quality-of-service agreements are mostly contracts in natural language which are not explicitly represented in the architecture or data of the system; their fulfillment is often monitored outside the usual system operation.

Current research in these areas is directed towards formal foundations and a more semantic understanding on the part of the systems so that declarative specifications of the desired system behavior, service-level or quality of service can be used to automatically negotiate the necessary contracts with possible partners and to create an orchestration of the system that fulfills the requirements.

The changing network structure described in Section 2.3, with devices which are not always connected to a network, emphasizes the need to replicate data. Replication is of course also useful for well-known other reasons [144]. Often traditional database techniques are not particularly well-suited for the structure of modern software-intensive systems. For example, it is difficult to ensure ACID properties without relying on a central coordinator and in a network where partitioning is likely to occur. Therefore techniques such as relaxed transactions or optimistic replication [145] or are often used instead of their traditional counterparts. To ensure reliability of the resulting systems additional techniques have to be used to compensate for the weaker guarantees provided by the infrastructure, e.g., rollback-recovery [146], compensation [147,148,149], or measures that change the system structure, e.g., by distributing the computation to fault-tolerant and transactional agents [150], or by provisioning backend databases reactively [151]. Another aspect which can be improved by distributed operation is scalability to extremely large data sets; one example is Google's *Bigtable* [152].

Human-System Interaction. In current systems human-system interaction is often based on interaction with keyboard and mouse or touchscreen, color displays and the WIMP (window, icon, menu, pointing-device) paradigm. Current research and development efforts are directed to design simple, self-explaining user interfaces, often with support for multimedia components or gesture-based and multi-modal interaction [153,154]. Research prototypes are exploring areas such as natural language interaction and "disappearing computers," i.e., embedded devices which either have no visible user interface at all, or where the user-interface is integrated into a real-world item and the software is controlled by interaction with that item. This research also leads in the direction of systems with which the user can interact in natural language [155,156,157,158], or which use sensors to react to the environment and the user [159,160,161].

The inverse direction is also an active topic in the development of human-system interactions: interactive worlds, simulations of real environments and augmented reality. Currently, interactive multi-user environments are commercially available and used for tasks such as social networking, online conferences or training. A popular example is Second Life [162]. One of the issues currently being addressed by the providers and users of these communities as well as by academic research is the management of virtual identities and the virtual representation of real-world items.

More generally, the increasing pervasiveness of software-intensive systems raises questions about better models for the acceptance of technology by users [163] and long-term human-computer relationships [164].

One issue related to human-computer interaction is managing the quality of the user experience and providing context-aware user interfaces that adapt to the user's personality, emotional state, current activities and the social context [165, 166, 167, 158, 168]. This is a challenging and active research area which will probably become more influential in the development of future products, in particular consumer devices, in the foreseeable future. Other aspects in the user

experience are accessibility [169,170] and the impact that delays, e.g., caused by network congestion, have on the user experience [171].

One aspect of the user experience that is often neglected in traditional approaches is that the user interface ensures privacy and security of the user's data and that it clearly communicates the implication of user interactions for these areas. It may be possible that formal methods for the design and checking of user interfaces [172,173] can prove advantageous in this area.

The presentation of systems that consist of many independent, dynamically configured parts is another current challenge in the area of human-system interaction. Current technologies, such as portlets, provide relatively low-level integration, where each node can display its own data. The approaches to present a more unified user interface for dynamically orchestrated systems will probably depend on a semantic analysis of the composition and the available data.

2.4 Resilience, Adaptation and Emergence

We call resilience the ability of a system to recover rapidly from negative influences such as component failure. Adaptation is the ability of a system to react in a useful manner to situations and environments that were not explicitly foreseen at the time of its development or deployment. In the software engineering literature, emergent behavior is often defined as behavior that cannot be localized in one component of the system but that instead arises from the structure and interaction of several independent parts of the system. Under this definition, non-functional properties such as performance are emergent properties [174] for all but the simplest systems. However, some authors use the term "emergent properties" only for properties that negatively impact the work of a system in ways that its designers did not foresee. Outside the software engineering literature, the term "emergence" is often used to denote more specific properties; see [175] for a extensive overview and references to the literature.

Many researchers agree that adaptation and emergent behaviors will be key features of future software-intensive systems [176,3] that pervade all aspects of their design and operation. The Volume see [177] contains a variety of viewpoints and background information.

While we have beautiful and intricate mathematical theories of complex dynamic systems [178,179,180], our current understanding of the concepts and theories for designing adaptive systems, or the mechanisms for controlling and exploiting emergent behaviors are rudimentary at best. We cannot reliably predict emergent behaviors of systems, let alone design systems that exhibit desirable emergent behaviors; current software possesses few properties that can be described as truly adaptive. The current state of the art are systems with several different configurations which can switch configurations based on external or internal criteria [181] or rule-based systems that exhibit a limited amount of adaptation [182]. We have currently no means to predict or exploit the emergent behavior in complex systems, and current engineering practices therefore try to eliminate the possibilities for emergent behavior as far as possible.

Most research about adaptation is interdisciplinary with cooperation between biologists, economists, social scientists, physicists, mathematicians and computer scientists, and drawing on areas such as control theory, cybernetics, systems theory, complexity theory, catastrophe theory, etc. Many publications on the subject can be obtained from [183].

Research directions in computer science that might impact the development of future adaptive systems are for example

- context-awareness; [184,168];
- automated reasoning about ontologies [185];
- automated reasoning about real-world data and problems [186];
- machine learning, user profiling and profiling of other systems, e.g., learning-based content management, learning- and profile-based user-interface selection [187];
- using "ambient intelligence" [188];
- self-* properties, such as self-configuration, self-monitoring, self-healing, self-tuning [189].

Emergent behavior and adaptation also influences the design of the human-computer interaction as it will be necessary to provide users of the system with enough information to allow them to monitor or control emergence [190].

Understanding emergence and developing tools to control and exploit emergent behavior will be one of the great research challenges in the next years.

2.5 Assurance

We use the term *assurance* to denote binding commitments that a system makes about properties that it promises to maintain (or that the user reasonably expects the system to maintain).

Quality of Service/Experience. One area of assurance is the quality of service (QoS) and the quality of user experience that the system provides. This encompasses many possible properties, such as guaranteed availability, maximum response times for certain interactions, guaranteed delivery of content, etc. Most current QoS agreements are written in natural language and not available as precise specifications for systems. Researchers are currently developing methods to automatically negotiate QoS agreements, but current approaches are not yet suitable for most applications. Furthermore, this is an area which is intimately connected to legal and contractual requirements which are beyond the scope of this chapter.

On a technical level, QoS concerns topics such as scheduling resources [191], the development of algorithms to ensure required levels of QoS [192,193], or finding routes that satisfy certain QoS requirements [194]. On a more abstract level, formalisms to specify and verify QoS requirements [195,196] play an important role in current research. Related to the issue of quality of service are compliance [197,198] and service-level agreements. In particular the formalization of service-level agreements is an important current research topic [199].

The quality of experience that a system shows is closely connected to both its technical ability to provide the service that its users expect, and to its human-computer interface as discussed in Section 2.3.

Security and Trust. We use the following definitions inspired by [200]: a trusted system or component is one whose failure can break the security policy; a trustworthy system is one that won't fail. There are several approaches to security engineering, both formal and informal. However experience with existing systems shows, that trusted systems fail with alarming regularity; the number of security breaches of Web-based system has almost doubled in 2007 compared to 2006. The increasing importance of networked systems in e-commerce and business transactions means that further research in the development of secure systems has to be a priority. However, many of the security breaches have relied on implementation errors rather than on conceptual weaknesses of the security systems [201]. Therefore, further improvements in the implementation of trusted systems and system components will be an important ingredient in the development of trustworthy software-intensive systems. Two of the leading researchers in computer security go so far as to claim: "In the past decade, cryptography has done more to damage the security of digital systems than it has to enhance it. [...] the mathematics of cryptography is almost never the weakest link. The fundamentals of cryptography are important, but far more important is how those fundamentals are implemented and used." [202].

Increasing network connectivity increases both the perceived and actual risk for systems [203,204]. There is a lot of material available about network security threats, [205] is a good overview of the state of the art as of 2002. An introduction to secure web applications is [206]. A survey of network defense mechanisms is available as [207], socio-technical aspects of security are addressed in [200,201,208], among many other sources. An important aspect of security is the detection of and response to security threats [209].

The increasing distribution of systems leads to several security issues which are currently not sufficiently well understood. One such problem is the security of protected data in environments with untrusted components.

Key management [210,211], access control [212,213], and more generally identity management are important problems in distributed systems. Many current systems rely on user names and passwords to authenticate their users. But as many users have to use dozens of different systems each day the management of passwords becomes unwieldy, leading many users to use the same password for all systems. Therefore a single compromised system can give an attacker access to user accounts on many other systems. Hardware-based solutions, such as tokens, smart cards or biometric systems exist and are often deployed to secure government or cooperate networks, however they are not commonly used for other purposes, such as authenticating Web sites. Single-sign-on solutions are technically feasible and have been deployed for some time, but they suffer from several problems: a security breach in a single-sign-on system can have dramatic consequences for the users since all their accounts are compromised, and the service providers relying on the single-sign-on provider have to entrust data about all their users to another company. For further information see [200,214].

Another issue with single-sign-on is privacy: the authentication provider can observe all visits of users to protected sites and correlate this information.

Similar privacy concerns exist in the area of online payment, as the of credit cards or payment services such as PayPal is directly traceable to the person making the payment. Anonymous online cash is technically feasible, but not widely used. In general, approaches to issues such as privacy, anonymity, pseudonymity, unobservability and unlinkability exist in research but are not widely deployed.

Many attack vectors exploit the interaction between different systems and rely on mismatches in the security boundaries between systems to subvert security mechanisms [201]. One current research direction to overcome these problems is the use of bio-inspired models to improve the security of networks [215]. As mentioned in Section 2.1, the availability of quantum computers would allow attacks on public-key cryptosystems and on protocols based on them. However it would also enable the use of quantum cryptography which is provably secure against certain kinds of attack [61].

One currently unsolved problem is to ensure that trustworthiness can be observed by the user of the system. In general users and administrators of today's systems have no mechanism to identify whether their data is transmitted to other systems, whether confidential data is retained, etc. Similarly the user interface has to be designed in a way that it respects the privacy of the user [216] without making the system inconvenient to use. Research in this area will be related to the research on context-aware user interfaces described in Section 2.3. There are some mechanisms to ensure that software has not been tampered with, e.g., by verifying secure hashes of the binaries and system data, however experience shows that these systems can often be bypassed, either because of implementation errors in the security system itself or because flaws in the system which are not directly related to the security components can be used to replace the program together with the authenticating information.

Since security breaches can never be completely ruled out, the ability to recognize if security breaches have happened is important. Intrusion detection systems that attempt to monitor the whole system and recognize both intrusion attempts and successful intrusions into parts of the system are commercially available and widely deployed. Their use is currently mostly confined to larger networks and servers although intrusion detection systems for single PCs or workstations exist. We are not aware of intrusion detection systems for programmable embedded devices, except for devices powerful enough to run software for desktop computers or workstations.

From a practical point of view, many security problems remain in operating systems and applications, but recently some significant increases have been made regarding the security in widely deployed operating systems, for example the introduction of mandatory access controls (MACs) or address-space randomization in the Red Hat Linux distribution [217,218]. Various studies have examined the security of open-source versus closed-source software [219,220].

To summarize, while there exist many areas in which future research is needed to ensure system security, most observed security breaches seem to rely at least partly on implementation errors, either because the designer of the security system was not aware of the often subtle implications of design choices on the

quality of the security system, or because the chosen implementation techniques were not adequate to build secure systems.

3 Engineering

The engineering of software-intensive systems poses many challenges, mostly because the complexity of software-intensive systems. We have structured this part of this chapter into several topics:

- *Engineering processes:* the various process models for system- and software engineering.
- *Distribution, Heterogeneity and Reuse:* approaches to deal with the distributed nature of and the heterogeneous components invariably found in systems, and reuse of existing components and system parts.
- *Separation of concerns:* approaches to structure the often multi-dimensional requirements.
- *Tool and language support:* The growing complexity of today's systems cannot be handled without adequate languages to express the requirements, design and implementation, and without support from tools.
- *Engineering for Resilience, Adaptivity and Emergence:* engineering techniques for systems that are resilient to failures, that can adapt to or be adapted to unforeseen circumstances, and techniques for controlling emergent behavior.

3.1 Engineering Processes

In the process of engineering software-intensive systems one has to distinguish between two activities: system engineering and software engineering. Systems engineering [221,222] is concerned with development of the whole system, the result of systems engineering are documents that describe the system architecture and the distribution of the system functionality to individual components of the system. Various methods for systems engineering exist, a commonly used set of procedures for the system engineering process is codified in the IEEE standard 1220 [223]. Reports on the state-of-practice show, that many companies do not follow a defined systems engineering process [7].

Software engineering is concerned with the development of the software for system components; the result of software engineering are software artifacts. An overview of the state of the art for software engineering can be found in [174].

Both systems and software engineering have to identify often complex requirements. An overview of the current state of the art in this area is given in [7,224], information about requirements for security and trust can be found in [225]. The article [226] provides an experience report for modeling systems with high dependability requirements.

Unless they can reuse components from earlier projects or commercial off-the-shelf (COTS) components, developers of software-intensive systems are faced

with the co-development of hard- and software. Because of the longer times required to develop and produce the hardware the co-development is often driven by the hardware requirements, although the situation is starting to change [7]. Often the teams responsible for software and hardware development are not well integrated, leading to misunderstandings and impedance mismatches in the developed artifacts. Techniques such as model-driven systems development [227,228] and simulation [229] are used to address these issues. Important issues in the requirements process are determining and managing the interaction of different requirements [230], and tracing the requirements throughout the software development process.

In the following paragraphs we will focus mostly on software engineering. .

There is an ongoing discussion about which software development process models are best suited for particular circumstances, and how tools and languages should support the process. Currently two directions which might appear contradictory seem to be prevalent. One trend is the use of model based development, mostly based on the UML [231]. In industrial practice the models are often manually implemented, or development is manually continued after initial code has been generated from the models. A lot of research is currently undertaken towards approaches such as MDA [232,233] which use transformational techniques to generate the complete code of the system directly from models. For model-based approaches the conformance and consistency of different models and different views is an important issue under active research, e.g., [234], but for which, as yet, few practial solutions exist.

The opposing trend is the rise of agile approaches [235,236,237,238] which stress the importance of incremental design and implementation over "up-front modeling." These approaches have gained many supporters in recent years, and many companies have started to include agile methods into their development process. It is likely that both kinds of process models, model driven and agile, will be used in practice for the foreseeable future.

Many parts of systems can be seen as members of a "family of components" which is used in several systems but where customization is necessary for different uses. The software development side for such components is addressed by approaches such as software product lines [239,240,9] or software factories [241]. Product line engineering tries to build families of products as configurations or variations of a single model; applicable design and modeling techniques are, e.g., domain-driven design [242] and feature modeling.

At a lower level of abstraction, design patterns are a common method to reuse modeling or implementation abstractions. General references are [243] for implementation-centric patterns, and [244,245,246,247,248,249] for architectural design patterns. The "Pattern Languages of Program Design" (PLoP) conferences are conferences dedicated to design patterns. Patterns for particular domains such as sensor networks are also widely published and used [250].

Testing has always been an important part of the development process of software-intensive systems. An overview is given in [251]. In industrial practice, the importance of early testing of individual components (unit testing) has been

particularly stressed by agile approaches which rely on the coverage of their test suite to detect regressions during refactoring [235]. Important testing aspects for systems are methods to compose test suites and cost-effective regression testing [252].

Formal methods are increasingly being used in the development of software-intensive systems. Automated theorem provers such as ACL2 [253], PVS [254], Specware [255], Coq [256], and HOL [257] have successfully been used to verify parts of microprocessors [258], similarly model checkers such as Spin [259,260], SMV [261], and Blast [262] are finding increasing application in particular in the area of finding defects in concurrent software designs. Model checkers have also recently been used to improve the performance of test suites, e.g., by eliminating redundant test cases [263,264,265].

3.2 Distribution, Heterogeneity and Reuse

Software engineering has to adjust to the fact that typical software products are becoming more complex, more distributed and less tightly integrated. Component techniques and languages promise the re-use of architecture and components, thereby increasing the level of abstraction at which the software developer works. Several techniques have been proposed to increase reconfigurability and facilitate fault localization in systems [266], to simplify or automate the configuration of distributed systems [267] or for feature location [268,269].

To cope with the massive number of nodes in current and future systems new abstractions have been proposed that are inspired by biological or chemical metaphors [270,271], see also Sect. 2.1 on page 6.

Other approaches to handle the increasing distribution of systems are based on notions of actors [272,273,274] or agents [275,276,277]. Here current research includes, among many other topics, protocols for multi-agent interaction [278], and tropos to increase the variability [279], or security of agent-based systems [280].

In recent years services have become the dominant technology for distributed computing. [268] gives an overview of the currently used technologies for Web services, [281] is a more conceptual overview of the field. Active research on services is performed in many areas, such as negotiation [282], service-level-agreements [199], choreography and orchestration [283], or verification of service-oriented systems [284,285].

Two independent roadmaps for service-oriented computing have been developed independently: one by the NESSI technology platform [286] and the other one by the International Conference on Service Oriented Computing. A number of European and international research programs are currently investigating the field of service engineering, among them Bionets [287], Cascadas [288], Ontogrid [289], Sims [290], Sodium [291], Plastic [292], Sensoria [293], One [294], Astro [295], Aosd [296], Music [297], WS-Diamond [298], Secse [299], Infrawebs [300], Dip [301], Amigo [302], Ws2 [303], Esfors [304], S3ms [305], Trustcom [306], Athena [307] and Dedisys [308]. An introduction to many of the European projects and their goals can be found in the forthcoming [309].

3.3 Separation of Concerns

Many requirements or system functionalities, for example logging or transactions, cannot easily be encapsulated in a single component but rather represent functionality that spans many different parts of a system. These functionalities are called "cross-cutting concerns" [310]. Object-oriented or functional software development approaches provide no direct support for encapsulating these cross-cutting concerns, therefore several extensions have been proposed under the name aspect-oriented programming (AOP). The common theme of different approaches to AOP is that different concerns of the program are specified separately and then combined in a modular way. Introductory articles about AOP are [311,312].

The best-known approach to AOP is the AspectJ extension to Java [313]. AspectJ introduces several concepts that are not present in standard object-oriented languages: *join-points* are points in the execution of programs, *point-cuts* are collections of join-points, and *advice* are method-like constructs that can be attached to point-cuts. This allows the modular implementation of features such as logging or transactions which would normally be distributed throughout the application. Other approaches to aspect-orientated programming are, e.g., based on adaptive methods [314], composition filters [315,316], or on multi-dimensional separation of concerns [317,318]. Aspect-orientation is currently also introduced into modeling languages such as the UML, see, e.g., [319].

Aspect-orientation is closely related to the more general topics of computational reflection [320,321] and meta-object protocols [322]. With the increasing need for dynamic adaptation and modification, aspect-oriented approaches based on general reflection and metaobject-protocols [323,324] seem to be promising areas of research.

A problem similar to cross-cutting concerns is context-sensitive behavior. Recent research in this area is, e.g., ContextL [325,326] in the area of programming languages and modes for software architecture [181].

Most approaches that address separation of concerns perform non-local transformations of models or software. Therefore not all testing and verification techniques are applicable without modification. One recent research effort in the area of verification is incremental aspect model-checking [327].

Model-based development approaches have advocated the use of various different but related models to represent systems for a long time. Model-integrated development [328,329] is a model-based approach to software development that uses domain-specific models to represent different concerns. Viewpoints have been proposed in [330,331] for a similar purpose. Related to these approaches are approaches to ensure traceability [332] between different architectural views, and between models and programs.

3.4 Language and Tool Support

A trend toward variety can be observed in the area of programming languages. Whereas a decade ago most software was developed in C/C++ with scripting

languages such as Perl playing an important role in web applications, today C/C++ are still prominent but no longer ubiquitous. Statically typed languages with more modern features such as garbage collection and introspection, the most common being Java and C#, are heavily used, particularly in the development of enterprise applications but also in other areas, and some languages with more advanced type systems, such as Scala [333] or F# [334] seem to become more visible in mainstream publications. On the other hand, dynamically typed languages are currently also seeing a significant rise in popularity, often integrated into the Java or .Net platforms. We expect these trends to continue in the next years, with increasing effort expended on cross-language interoperability.

A number of research efforts try to enhance the expressiveness of program source code. One example is the area of domain-specific languages [335]: here the goal is to simplify the development of systems for particular areas by first defining a language in which the necessary concepts can be concisely expressed and then writing the application in this language. Another approach is to employ concern graphs [336] to document how the implementations of concerns are distributed throughout the source code.

In recent years modern languages have been more widely used in the development of real-time and embedded systems than previously. This was caused in part by the increasing computational power available on these devices, but also by research efforts to provide support for real-time systems in the run-time of modern languages, e.g., by providing real-time garbage collection [337]. Novel methods for programming massively distributed systems are currently under investigation [338,339].

For many systems the notion of shutting them down to perform upgrades or maintenance is no longer feasible. This is not a completely new phenomenon, since for example, the power grid has to continue working, even if parts of its infrastructure are updated. However the increasing number of large-scale, distributed systems will make this situation more common. Most current tools are not well suited to develop for systems in which no distinction between development- and run-time exist, and not even for systems where individual parts can be replaced at run-time, although research and some industrial solutions exist [340].

4 Research Challenges

After reviewing the state of the art of engineering software-intensive systems, we now take a look into the future: What are the research challenges facing us in this area? The following is derived from the results of the workshops of the InterLink Working Group 1.

Many of the numerous challenges that we expect to see in the development of software-intensive systems can be classified as belonging to one of the following areas: massive numbers of nodes per system, open and non-deterministic environments, and adaptation:

Massive Numbers of Nodes per System. A massive increase in the number of nodes of future software-intensive systems will be one of the most

visible features: the development of multi-core processors with tens to thousands of cores integrated on a single chip implies that even single-chip systems will have to be treated as consisting of large numbers of individual nodes, the availability of cheap, low-energy mobile devices ensures that we will see an increasing number of elements with computational capability in the next years. Both of these trends are true even for systems built out of traditional computing devices. Furthermore, new manufacturing methods such as nanotechnology, synthetic biology, or MEMS will give rise to new kinds of ensembles, many of them with potentially millions of computational nodes.

Open and Non-Deterministic Environments. Currently many PDAs, mobile phones, personal computers, workstations and servers used in commercial environments are networked—via local area networks, dedicated wide area networks, or globally via the Internet. We are currently seeing this trend for many other devices, such as embedded systems, or even RFID-equipped consumer goods, as well. The increase in available devices in turn entices service providers to offer new services or to remove service offers that are no longer profitable or that have been superseded by newer offers. Therefore, software-intensive systems can no longer expect to operate in the environment that was current during their design time—they have to replace services that are no longer available with others that offer similar functionality; they should also take advantage of new services provided by the network environment that were not foreseen when the system was designed.

Adaptation. The situation mentioned in the previous paragraph is one instance of a more general situation: future systems will often have to operate under conditions that differ significantly from the ones for which they were designed. They should not only be able to adapt to changes in their network environment, they should also be able to work reliably in the face of changes to their execution platform: even today reinstalling all necessary programs is a major burden when we switch to a new computer. Since future software-intensive systems will be more ubiquitous, more distributed, and will assimilate a large number of adaptations during their operation, the prospect of reinstalling them from scratch when switching to a new platform becomes infeasible. Therefore we need to develop systems that can adapt to different environments, to different users, etc.

We call this future generation of software-intensive systems *ensembles*. *Ensemble engineering* is the science and engineering discipline of complex, integrated ensembles of computing elements. The huge impact—both positive and negative—that ensembles will have on society means that we need to understand ways to reliably and predictably model, design, and program them.

Ensemble engineering is closely connected to, and incorporates ideas from, other research areas such as software engineering, artificial intelligence, complex adaptive systems and non-conventional programming paradigms. Ensembles can be categorized along several different axes, for example:

- Physical—Virtual: Is the system mainly concerned with sensing and acting in the real world, or is it purely virtual, or is it somewhere in between?
- Homogenous—Inhomogenous: Is the system made of (large numbers of) identical parts, is it made of (a large number of) different components taken from a few different types of component, or does it consist of a large number of parts, all of which are different.
- Cooperative—Competitive: Do the individual parts cooperate to achieve a certain task or do they compete with each other. There might also be systems which are in-between, such as a system where the components are cooperating but competing for a scarce resource to fulfill their individual tasks.

Further distinctions between systems are the topology of their communication links, the average computational power of the nodes, or the purpose of the system, etc. While some research topics are specific to a certain kind of ensemble or a certain application area, many are common to all ensembles.

4.1 Research Challenges for Ensemble Engineering

In the InterLink WG1 workshops, the following long-term research challenges have been identified. They can be classified as one of the categories "properties of ensembles", "specification and design", "assurances", and "implementation and verification".

Properties of Ensembles. The increase in scale along a variety of dimensions leads to ensembles having either new unique properties or displaying properties of traditional systems, but on a much larger scale.

- *Harnessing the stochastic behavior and massive scale.* There are several reasons for stochastic behavior in ensembles: Large numbers of independent parallel devices can usually not be synchronized without creating inefficiencies; the resulting behavior can often only be described in a stochastic manner because of the huge number of combinatorical possibilities. Furthermore, massive numbers of components cause high probabilities for failures of individual components, even if the probability of failure for any individual component is relatively low. This effect is compounded by techniques such as nano-technology which inheerently have high component defect rates.
  With current development methods both stochastic behavior and massive scale pose problems for system development. This does not necessarily have to be the case, as can be seen in biological systems which harness these effects in order to achieve resilience and adaptation. We need to find ways to do the same in software-intensive systems as well.
- *Resilience, adaptation, and controlled emergence.* Today's systems are overly brittle: a single failure in a typical computer program will cause it to "crash," failure of single components impede the functionality of the overall system, security breaches leave whole systems exposed to attackers. We have to develop methods to build resilient systems, that can cope with component faliures or security breaches in individual components without degrading

the overall system performance or security; we have to build systems that can adapt to changing situations or requirements without significant human intervention or redeployment; we have to control emergent behaviors and harness the positive aspects of emergence.
- *Operating in open environments, recognizing and exploiting opportunities.* This challenge repeats a point that has already been mentioned several times: dynamicity and openness of environments present significant obstacles to system performance and reliability. We need to find ways to reverse this situation and build systems that can recognize and exploit opportunities arising in these circumstances.
- *Designing and predicting emergent behavior.* Many of the behaviors of ensembles result not from actions of a single entity but rather from the combined activities of several independent nodes, either in a pre-planned or in a spontaneous manner. We need to better understand the behavior of concurrent systems, so that we can design and predict these emergent behaviors.
- *Mobility of code, data, and devices.* In some situations, mobility of code or data can be used to increase the responsiveness, performance or reliability of systems, or to work around system limitations, such as low network bandwith. Mobility of devices leads to a dynamically changing network topology and the need for services to adapt to the continuous appearance and disappearance of nodes in a system.
- *Integration of heterogeneous and federated data.* With large systems, distributed across several organizational boundaries and integrating parts from different vendors, we can no longer expect the system to share a single, homogenous data model. Instead we have to find ways to integrate heterogeneous, federated, and mutually inconsistent data sources that invariably form part of large systems.

Specification and Design. The emergent behavior and open-ended nature of ensembles poses new challenges for specification and design.

- *Deducing a global specification from local rules, and finding local rules that produce a desired global behavior. Influencing the global behavior by making local changes.* It will often be no longer possible to design the complete system according to a certain specification; instead we will have to integrate services as parts into a larger environment. We will thus no longer be able to use a top-down approach to specify the system behavior. Instead we will have to develop methods for influencing the global system behavior by affecting only the local environment of individual components, not the global system behavior.
- *Abstractions and models for massively parallel, open-ended systems.* We currently lack the necessary abstractions and foundations to describe, model, and design massively parallel systems. New theoretical foundations and abstractions are urgently needed.

Assurances. Ensembles being both adaptive and being used in safety-critical systems makes it a necessity that one can give assurances regarding their performance.

- *Replacing the notion of correctness with the one of "fitness for a purpose."* For systems operating in complex, dynamically changing, open-ended environments it is often no longer useful or even possible to specify a notion of correctness. Instead we need to design systems that are "fit for a particular purpose" and can achieve their desired results in varying and changing situations.
- *Social and emotional perception, quality of experience.* Current human-computer interactions are mostly influenced by the limitations of the computer: it is neither aware of the changing social context in which an interaction may take place, nor of the emotional or physical state of its users. As software-intensive systems pervade ever increasing parts of our professional and private lives, the interaction has to be focused more on the quality of experience for the human users.
- *Assurance guarantees, certification, and formal verification of ensembles.* Future systems, that adapt to their environment, exploit new oportunities, and take into account their user's emotional state should still provide performance guarantees and limits (e.g., not harming people). We need to develop ways to certify these systems and to formally verify their properties.
- *Privacy, identity management, security, and trust.* As the importance of our online presence and identity have increased the problems of privacy, identity management, security and trust have become more apparent—althogh often only because of negative consequences that received media attention. For future systems we need to develop methods and tools to ensure these properties, and techniques to reliably communicate the security aspects of actions to users.
- *Quality of service, quality of experience.* To build reliable systems, components need to be able to negotiate quality of service criteria, preferably without manual intervention, and then be able to perform according to these criteria. On a higher level, many systems need to provide a certain "quality of experience" for their users.

Implementation and Verification. If we are to ever build ensembles with the high standards we have set so far, we will need tools for doing so. These tools will be both evolutionary improvements of current tools, but will also need to incorporate revoluationary ideas to enable the above mentioned goals.

- *Programming languages, development environments, and compiler technology for ensembles.* Current programming languages, development environments and compilers are not adequate for the development of long-running, adaptive systems. For example, they lack features for monitoring, debugging, patching, or upgrading deployed systems from the development environment, or for upgrading systems while they are operating. They also offer no support for dealing with distributed systems which are only locally consistent, a situation which arises frequently in ensembles.
- *Testing and verification of ensembles.* Many of the complexities in the development of ensembles arise from the interactions of concurrently executing,

(semi-)autonomous nodes. While test and verification methods have greatly improved in the last years, they are still not sufficient for current or future systems. In particular, verification of large or open systems are still unsolved problems.
- *Multi-paradigm programming.* In current software systems, several formal languages exist next to each other: General-purpose languages such as Java, database languages such as SQL, specification languages, etc. These languages adhere to different paradigms which means that development environments need to support *several* paradigms. Furthermore, diversity in programming languages will always exist (as there is too much knowledge encoded in legacy source code). Thus, one needs to support legacy paradigms *in addition to* new break-through paradigms.
- *Dynamism in programming languages.* Dynamism in programs appears in several facets: Boundaries between programs dissolve, as integration will be much more fine-grained. Formerly clearly separated software lifecycle stages (testing, deployment, updates) are becoming more tightly interlocked. Furthermore, self-* properties, especially where humans are concerned introduce additional levels of reactivity and dynamism. The challenge is to provide adequate tools and methods for dealing with this kind of dynamism: How can their design and implementation be supported? What kind of assurances can we give?

4.2 Research Areas

The discussions of the InterLink WG1 have focused on three research areas:

- *Physical ensembles,* which are intimately connected to the physical world in space and time. They are equipped with sensors and actuators and have to take into account issues of locality and resource constraints. Examples are real-time embedded systems, claytronics, modular robots or programmable matter. These systems combine discrete and continuous, non-linear domains, and they exhibit complex interaction patterns between components. Coordination in space and time with limited resources is one of the major challenges faced by physical ensembles.
- *Organic software and systems engineering* addresses the challenge of designing software for ensembles that is reliable, predictable, with guarantees for security and trust, that acts autonomously and has self-* properties, and harnesses emergent behavior. Approaches to organic software engineering may use bio-inspired and swarm algorithms and rely on nature-inspired programming paradigms, but they also include traditional software development techniques and formal methods.
- *Societal computing* addresses the problem of composing "living" and evolving societal software systems from parts that were not designed to be composed together and which may partially compete with each other while partially cooperating. These systems will require languages and environments which blur the distinction between compile-time and run-time and between design,

implementation and deployment. Research in this area will have to investigate the dynamics of purposive interactions and the structure of evolving societal architectures: evolution of societal software systems has to be a long-term process that goes beyond single-run adaptation, and systems have to maintain societal coherence while supporting diversity and context awareness.

5 Conclusion

Software-intensive systems play an important role in almost every part of human society: Transportation systems, communication systems, energy networks, medical technology, etc. That is, they exist *today*, but they are very difficult to construct. This document gave an overview of how software-intensive systems are currently built. It then describes areas where software-intensive systems will face even greater challenges in the future: The number of nodes per system will increase, systems will be deployed in environments that are more and more open and non-deterministic, and adaptation in these environments will be essential. Software-intensive systems to which this characterization applies have been called *ensembles* by the InterLink WG1. This document described areas of research that might help with meeting the challenges of engineering ensembles. That ensembles will be crucial for social and economic competetiveness in the future is virtually certain, it is now upon the research community to help build, harness, and understand them.

Acknowledgements

This work was partially funded by the IST Coordinated Action InterLink "International Coordination Activities in Future and Emerging ICTs" (034051) and the IST Integrated Project Sensoria, IST-2 005-016004. Without the input of the members of the InterLink WG1 "Software-Intensive Systems and New Computing Paradigms" this chapter would not have been possible and we gratefully acknowledge their contribution.

References

1. Dedrick, J., Gurbaxani, V., Kraemer, K.L.: Information technology and economic performance: A critical review of the empirical evidence. ACM Comput. Surv. 35(1), 1–28 (2003)
2. Information Technology for European Advancement (ITEA) Office Association: ITEA technology roadmap for software-intensive systems, 2nd edn. (May 2004), http://www.itea-office.org
3. Wirsing, M., Hölzl, M.: Software-intensive systems. Report of the Beyond-the-Horizon WG6 (2007)
4. Glass, R.L.: Guest editor's introduction: The state of the practice of software engineering. IEEE Software 20(6), 20–21 (2003)

5. Cusumano, M.A., MacCormack, A., Kemerer, C.F., Crandall, B.: Software development worldwide: The state of the practice. IEEE Software 20(6), 28–34 (2003)
6. Lethbridge, T., Singer, J., Forward, A.: How software engineers use documentation: The state of the practice. IEEE Software 20(6), 35–39 (2003)
7. Neill, C.J., Laplante, P.A.: Requirements engineering: The state of the practice. IEEE Software 20(6), 40–45 (2003)
8. Ciolkowski, M., Laitenberger, O., Biffl, S.: Software reviews: The state of the practice. IEEE Software 20(6), 46–51 (2003)
9. Birk, A., Heller, G., John, I., Schmid, K., von der Maßen, T., Müller, K.: Product line engineering: The state of the practice. IEEE Software 20(6), 52–60 (2003)
10. Graaf, B., Lormans, M., Toetenel, H.: Embedded software engineering: The state of the practice. IEEE Software 20(6), 61–69 (2003)
11. Baskerville, R., Ramesh, B., Levine, L., Pries-Heje, J., Slaughter, S.: Is internet-speed software development different? IEEE Software 20(6), 70–77 (2003)
12. Reifer, D.J.: Is the software engineering state of the practice getting closer to the state of the art? IEEE Software 20(6), 78–83 (2003)
13. Moore, G.E.: Cramming more components onto integrated circuits. Electronics 38(8), 114–117 (1965)
14. Kopp, C.: Moore's Law and its Implications for Information Warfare. In: Sibilia, R. (ed.) Proceedings of the International AOC Electronic Warfare Conference, Alexandria, Virginia, USA, May 20-25, 2000, vol. 3, Association of Old Crows, AOC International AOC Electronic Warfare Conference (2000)
15. Coatta, T.: The (not so) hidden computer. Queue 4(3), 22–26 (2006)
16. Benetti, S.: Intelligent co-operative systems in cars for road safety (July 2006)
17. Lee, E.A.: Cyber-physical systems—are computing foundations adequate? In: NSF Workshop on Cyber-Physical Systems (October 2006)
18. NSF Workshop on Cyber-Physical Systems: Web site of the nsf workshop on cyber-physical systems (October 2006) (last accessed 2008-01-20), http://varma.ece.cmu.edu/cps/
19. Ungerer, T., Robič, B., Šilc, J.: A survey of processors with explicit multithreading. ACM Comput. Surv. 35(1), 29–63 (2003)
20. Bjerregaard, T., Mahadevan, S.: A survey of research and practices of network-on-chip. ACM Comput. Surv. 38(1), 1 (2006)
21. Lyshevski, S.E.: MEMS and NEMS: Systems, Devices and Structures. CRC Press, Boca Raton (2002)
22. Gad-El-Hak, M. (ed.): The MEMS Handbook. CRC Press, Boca Raton (2001)
23. Allen, J.J.: Micro Electro Mechanical System Design. CRC Press, Boca Raton (2005)
24. Gad-El-Hak, M.: MEMS—Introduction and Fundamentals. CRC Press, Boca Raton (2006)
25. Memsnet: Web site of the memsnet.org project (2008) (last accessed 2008-01-10), http://www.memsnet.org/
26. Venkatachalam, V., Franz, M.: Power reduction techniques for microprocessor systems. ACM Comput. Surv. 37(3), 195–237 (2005)
27. Rad, R.M., Tehranipoor, M.: Evaluating area and performance of hybrid fpgas with nanoscale clusters and cmos routing. J. Emerg. Technol. Comput. Syst. 3(3), 15 (2007)
28. Massoud, Y., Nieuwoudt, A.: Modeling and design challenges and solutions for carbon nanotube-based interconnect in future high performance integrated circuits. J. Emerg. Technol. Comput. Syst. 2(3), 155–196 (2006)

29. Compton, K., Hauck, S.: Reconfigurable computing: a survey of systems and software. ACM Comput. Surv. 34(2), 171–210 (2002)
30. Goldstein, S.C., Mowry, T.C.: Claytronics: A scalable basis for future robots. In: RoboSphere 2004, Moffett Field, CA (November 2004)
31. Kirby, B., Aksak, B., Goldstein, S.C., Hoburg, J.F., Mowry, T.C., Pillai, P.: A modular robotic system using magnetic force effectors. In: Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS 2007) (October 2007)
32. Ravichandran, R., Gordon, G., Goldstein, S.C.: A scalable distributed algorithm for shape transformation in multi-robot systems. In: Proceedings of the IEEE International Conference on Intelligent Robots and Systems IROS 2007 (October 2007)
33. Tahoori, M.B.: Application-independent defect tolerance of reconfigurable nanoarchitectures. J. Emerg. Technol. Comput. Syst. 2(3), 197–218 (2006)
34. Prodan, L., Udrescu, M., Boncalo, O., Vladutiu, M.: Design for dependability in emerging technologies. J. Emerg. Technol. Comput. Syst. 3(2), 6 (2007)
35. Sekanina, L.: Evolutionary functional recovery in virtual reconfigurable circuits. J. Emerg. Technol. Comput. Syst. 3(2), 8 (2007)
36. Tempesti, G., Mange, D., Mudry, P.A., Rossier, J., Stauffer, A.: Self-replicating hardware for reliability: The embryonics project. J. Emerg. Technol. Comput. Syst. 3(2), 9 (2007)
37. Patwardhan, J., Dwyer, C., Lebeck, A.R.: A self-organizing defect tolerant simd architecture. J. Emerg. Technol. Comput. Syst. 3(2), 10 (2007)
38. Milner, R., Stepney, S.: Nanotechnology: Computer science opportunities and challenges, Technical report, Submission by the UK Computing Research Committee to the Nanotechnology Working Group of the Royal Society and the Royal Academy of Engineering (August 2003)
39. Stepney, S., Braunstein, S.L., Clark, J.A., Tyrrell, A.M., Adamatzky, A., Smith, R.E., Addis, T.R., Johnson, C.G., Timmis, J., Welch, P.H., Milner, R., Partridge, D.: Journeys in non-classical computation II: initial journeys and waypoints. Parallel Algorithms Appl 21(2), 97–125 (2006)
40. Feynman, R.P.: There's plenty of room at the bottom—an invitation to enter a new field of physics. Engineering and Science (February 1960)
41. Drexler, K.E.: Molecular engineering: An approach to the development of general capabilities for molecular manipulation. Proc. Nat. Acad. Sci. USA 78(9), 5275–5278 (1981)
42. Aaron, R., Herr, H., Ciombor, D., Hochberg, L., Donoghue, J., Briant, C., Morgan, J., Ehrlich, M.: Horizons in prosthesis development for the restoration of limb function. Journal of the American Academy of Orthopaedic Surgeons 14(10), 198–204 (2006)
43. Su, F., Chakrabarty, K.: Yield enhancement of reconfigurable microfluidics-based biochips using interstitial redundancy. J. Emerg. Technol. Comput. Syst. 2(2), 104–128 (2006)
44. Amos, M.: Theoretical and Experimental DNA Computation. Springer, Heidelberg (2005)
45. Head, T.: Formal language theory and dna: an analysis of the generative capacity of specific recombinant behaviors. Bull. Math. Biology 49(6), 737–759 (1987)
46. Adleman, L.: Molecular computation of solutions to combinatorial problems. Science 266(5187), 1021–1024 (1994)
47. Hogg, T., Huberman, B.A.: Controlling smart matter. Smart Materials and Structures 7(R1) (1998)

48. Maclennan, B.J.: Replication, sharing, deletion, lists, and numerals: Progress on universally programmable intelligent matter (November 2002)
49. Rothemund, P.W.K., Papadakis, N., Winfree, E.: Algorithmic self-assembly of DNA Sierpinski triangles. In: Preliminary Proceedings of DNA Computing, 9th international Workshop on DNA-Based Computers, DNA 2003, 125 Madison, Wisconsin, USA, pp. 1–4 (June 2003)
50. Ben-Hur, A., Siegelmann, H.T.: Computation in gene networks. Chaos 14(1), 145–151 (2004)
51. Kobayashi, H., Kaern, M., Araki, M., Chung, K., Gardner, T.S., Cantor, C.R., Collins, J.J.: Programmable cells: interfacing natural and engineered gene networks. Proc. Natl. Acad. Sci. U S A 101(22), 8414–8419 (2004)
52. Seelig, G., Soloveichik, D., Zhang, D.Y., Winfree, E.: Enzyme-free nucleic acid logic circuits. Science 314(5805), 1585–1588 (2006)
53. Yokobayashi, Y., Weiss, R., Arnold, F.H.: Directed evolution of a genetic circuit. Proc. Natl. Acad. Sci. U S A 99(26), 16587–16591 (2002)
54. Franois, P., Hakim, V.: Design of genetic networks with specified functions by evolution in silicon. Proc. Natl. Acad. Sci. U S A 101(2), 580–585 (2004)
55. Weiss, R., Knight, T.: Engineered communications for microbial robotics. In: DNA: International Workshop on DNA-Based Computers. LNCS. Springer, Heidelberg (2000)
56. Weiss, R., Basu, S., Hooshangi, S., Kalmbach, A., Karig, D., Mehreja, R., Netravali, I.: Genetic circuit building blocks for cellular computation, communications, and signal processing. Natural Computing 2(1), 43–84 (2003)
57. Gibbs, W.W.: Synthetic life. Scientific American 290(5), 74–81 (2004)
58. Syntheticbiology.org: Syntheticbiology.org Web site (2007), http://syntheticbiology.org/
59. GEM: Web site of the international Genetically Engineered Machine competition (2007), http://parts.mit.edu/wiki
60. Carlson, R.: The pace and proliferation of biological technologies. Biosecur Bioterror 1(3), 203–214 (2003)
61. Bruss, D., Erdélyi, G., Meyer, T., Riege, T., Rothe, J.: Quantum cryptography: A survey. ACM Comput. Surv. 39(2), 6 (2007)
62. Van der Pyl, T., Karlson, A. (eds.): Quantum Information Processing & Communications in Europe. European Union, IST-FET (2005)
63. ERA-Pilot: Web site of the ERA-Pilot QIST Coordinated Action (2008) (last accessed 2008-01-07), http://www.qist-europe.net/
64. Meter, R.V., Oskin, M.: Architectural implications of quantum computing technologies. J. Emerg. Technol. Comput. Syst. 2(1), 31–63 (2006)
65. Naur, P.: Computing versus human thinking. Commun. ACM 50(1), 85–94 (2007)
66. Yilmaz, A., Javed, O., Shah, M.: Object tracking: A survey. ACM Comput. Surv. 38(4), 13 (2006)
67. Lobo, J., Dias, J.: Fusing of image and inertial sensing for camera calibration. In: Proceedings of the IEEE Conference on Multisensor Fusion and Integration for Intelligent Systems, MFI (2000)
68. Clouqueur, T., Phipatanasuphorn, V., Ramanathan, P., Saluja, K.: Sensor deployment strategy for target detection. In: The First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA 2002) (September 2002)
69. Lyons, D.M., Hsu, D.F., Ma, Q., Wang, L.: Combinatorial fusion criteria for robot mapping. In: AINA, pp. 847–852. IEEE Computer Society, Los Alamitos (2007)

70. Gechter, F., Chevrier, V., Charpillet, F.: A reactive agent-based problem-solving model: Application to localization and tracking. ACM Trans. Auton. Adapt. Syst. 1(2), 189–222 (2006)
71. Nakamura, E.F., Loureiro, A.A.F., Frery, A.C.: Information fusion for wireless sensor networks: Methods, models, and classifications. ACM Comput. Surv. 39(3), 9 (2007)
72. Ramachandran, U., Kumar, R., Wolenetz, M., Cooper, B., Agarwalla, B., Shin, J., Hutto, P., Paul, A.: Dynamic data fusion for future sensor networks. ACM Trans. Sen. Netw. 2(3), 404–443 (2006)
73. Ganesan, D., Cristescu, R., Beferull-Lozano, B.: Power-efficient sensor placement and transmission structure for data gathering under distortion constraints. ACM Trans. Sen. Netw. 2(2), 155–181 (2006)
74. Zhou, G., He, T., Krishnamurthy, S., Stankovic, J.A.: Models and solutions for radio irregularity in wireless sensor networks. ACM Trans. Sen. Netw. 2(2), 221–262 (2006)
75. Kansal, A., Hsu, J., Zahedi, S., Srivastava, M.B.: Power management in energy harvesting sensor networks. Trans. on Embedded Computing Sys. 6(4), 32 (2007)
76. Chakrabarti, A., Sabharwal, A., Aazhang, B.: Communication power optimization in a sensor network with a path-constrained mobile observer. ACM Trans. Sen. Netw. 2(3), 297–324 (2006)
77. Kansal, A., Kaiser, W., Pottie, G., Srivastava, M., Sukhatme, G.: Reconfiguration methods for mobile sensor networks. ACM Trans. Sen. Netw. 3(4), 22 (2007)
78. Karnik, A., Kumar, A.: Distributed optimal self-organization in ad hoc wireless sensor networks. IEEE/ACM Trans. Netw. 15(5), 1035–1045 (2007)
79. Huang, C.F., Tseng, Y.C., Wu, H.L.: Distributed protocols for ensuring both coverage and connectivity of a wireless sensor network. ACM Trans. Sen. Netw. 3(1), 5 (2007)
80. Lazos, L., Poovendran, R.: Stochastic coverage in heterogeneous sensor networks. ACM Trans. Sen. Netw. 2(3), 325–358 (2006)
81. Cărbunar, B., Grama, A., Vitek, J., Cărbunar, O.: Redundancy and coverage detection in sensor networks. ACM Trans. Sen. Netw. 2(1), 94–128 (2006)
82. Yoon, S., Veerarittiphan, C., Sichitiu, M.L.: Tiny-sync: Tight time synchronization for wireless sensor networks. ACM Trans. Sen. Netw. 3(2), 8 (2007)
83. Farrugia, E., Simon, R.: An efficient and secure protocol for sensor network time synchronization. Journal of Systems and Software 79(2), 147–162 (2006)
84. Lanese, I.: Synchronization Strategies for Global Computing Models. PhD thesis, Ph.D. school in Computer Science, University of Pisa, Pisa, Italy (2006)
85. Jindal, A., Psounis, K.: Modeling spatially correlated data in sensor networks. ACM Trans. Sen. Netw. 2(4), 466–499 (2006)
86. Herbert, D., Sundaram, V., Lu, Y.H., Bagchi, S., Li, Z.: Adaptive correctness monitoring for wireless sensor networks using hierarchical distributed run-time invariant checking. ACM Trans. Auton. Adapt. Syst. 2(3), 8 (2007)
87. Wang, D., Zhang, Q., Liu, J.: The self-protection problem in wireless sensor networks. ACM Trans. Sen. Netw. 3(4), 20 (2007)
88. Salatian, A., Hunter, J.: Deriving trends in historical and real-time continuously sampled medical data. J. Intell. Inf. Syst. 13(1-2), 47–71 (1999)
89. Combi, C., Chittaro, L.: Abstraction on clinical data sequences: an object-oriented data model and a query language based on the event calculus. Artificial Intelligence in Medicine 17(3), 271–301 (1999)

90. Matuszek, C., Witbrock, M.J., Kahlert, R.C., Cabral, J., Schneider, D., Shah, P., Lenat, D.B.: Searching for common sense: Populating cyc from the web. In: Veloso, M.M., Kambhampati, S. (eds.) AAAI, pp. 1430–1435. AAAI Press / The MIT Press (2005)
91. Taylor, M.E., Matuszek, C., Klimt, B., Witbrock, M.J.: Autonomous classification of knowledge into an ontology. In: Wilson, D., Sutcliffe, G. (eds.) FLAIRS Conference, pp. 140–145. AAAI Press, Menlo Park (2007)
92. NASA: Intelligent data understanding subproject of the intelligent systems project (2008), http://ti.arc.nasa.gov/is/IDU/index.html
93. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F., Cowan, J.: Extensible markup language (xml) 1.1, 2nd edn. W3C Recommendation (August 2006), http://www.w3.org/TR/2006/REC-xml11-20060816/
94. W3C Consortium: Extensible markup language (xml) web site (2008) (last accessed: 2008-01-19), http://www.w3.org/XML/
95. Object Management Group: Meta Object Facility (MOF) Specification Version 1.4 (April 2002) OMG Document: formal/02-04-03, http://www.omg.org/docs/formal/02-04-03.pdf
96. W3C Consortium: Resource description framework (rdf) web site (2008) (last accessed: 2008-01-19), http://www.w3.org/RDF/
97. KIF: Knowledge interchange format—draft proposed american national standard (dpans) NCITS.T2/98-004 (1998), http://logic.stanford.edu/kif/dpans.html
98. Lenat, D.B., Guha, R.V.: The evolution of cycl, the cyc representation language. SIGART Bull. 2(3), 84–87 (1991)
99. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications, 2nd edn. Cambridge University Press, Cambridge (2007)
100. McGuinness, D.L., van Harmelen, F.: Owl web ontology language: Overview. W3C Recommendation, http://www.w3.org/TR/owl-features/
101. Bechhofer, S., et al.: Owl web ontology language: Reference. W3C Recommendation, http://www.w3.org/TR/owl-ref/
102. Berners-Lee, T., Connolly, D., Kagal, L., Scharf, Y., Hendler, J.: N3logic: A logical framework for the world wide web (2007), http://arxiv.org/abs/0711.1533
103. Altheim, M., Anderson, B., Hayes, P., Menzel, C., Sowa, J.F., Tammet, T.: Scl: Simple common logic (2004), http://www.ihmc.us/users/phayes/CL/SCL2004.html
104. Group, O.M.: Ontology definition metamodel (OMG adopted specification) OMG Document Number: ptc/2007-09-09 (2007), http://www.omg.org/cgi-bin/doc?ptc/07-09-09
105. Copeland, J.B.: Cyc: A case study in ontological engineering. Electronic Journal of Analytic Philosophy 5 (1997)
106. Cyc: The Cyc foundation. http://www.cycfoundation.org/
107. Fellbaum, C. (ed.): WordNet—An Electronic Lexical Database. MIT Press, Cambridge (1998)
108. Niles, I., Pease, A.: Towards a standard upper ontology. In: Proceedings of the 2nd International Conference on Formal Ontology in Information Systems, FOIS 2001 (2001)
109. Formica, A., Missikoff, M.: Inheritance processing and conflicts in structural generalization hierarchies. ACM Comput. Surv. 36(3), 263–290 (2004)
110. Katifori, A., Halatsis, C., Lepouras, G., Vassilakis, C., Giannopoulou, E.: Ontology visualization methods—a survey. ACM Comput. Surv. 39(4), 10 (2007)

111. Motik, B., Stojanovic, N., Stojanovic, L., Maedche, A.: User-driven ontology evolution management. In: Gómez-Pérez, A., Benjamins, V.R. (eds.) EKAW 2002. LNCS (LNAI), vol. 2473. Springer, Heidelberg (2002)
112. Klein, M., Fensel, D.: Ontology versioning on the semantic web. In: Int. Semantic Web Working Symp. (SWWS) (2001)
113. Wikipedia: Rdfa, http://en.wikipedia.org/wiki/RDFa
114. Bernstein, A., Kaufmann, E., Göhring, A., Kiefer, C.: Querying ontologies: A controlled english interface for end-users. In: 4th Int. Semantic Web Conf. (ISWC) (2005)
115. Kuhn, T.: AceWiki: Collaborative Ontology Management in Controlled Natural Language. In: Proc. 3rd Semantic Wiki Wsh. CEUR Workshop Proceedings (2008)
116. Benjamins, V.R., Davies, J., Baeza-Yates, R., Mika, P., Zaragoza, H., Greaves, M., Gmez-Prez, J.M., Contreras, J., Domingue, J., Fensel, D.: Near-term prospects for semantic technologies. IEEE Intelligent Systems 23(1), 76–88 (2008)
117. Alferes, J.J., Bailey, J., May, W., Schwertel, U. (eds.): PPSWR 2006. LNCS, vol. 4187. Springer, Heidelberg (2006)
118. Wyss, C.M., James, A., Hasselbring, W., Conrad, S., Höpfner, H.: Report on the engineering federated information systems 2003 workshop (efis 2003). SIGSOFT Softw. Eng. Notes 29(2), 1–3 (2004)
119. Sohn, D.: Understanding drm. Queue 5(7), 32–39 (2007)
120. Zheng, D., Liu, Y., Zhao, J., Saddik, A.E.: A survey of rst invariant image watermarking algorithms. ACM Comput. 39(2), 5 (2007)
121. Garfinkel, S.: Database Nation: The Death of Privacy in the 21st Century. O'Reilly, Sebastopol (2000)
122. Turmo, J., Ageno, A., Català, N.: Adaptive information extraction. ACM Comput. Surv. 38(2), 4 (2006)
123. Ceglar, A., Roddick, J.F.: Association mining. ACM Comput. Surv. 38(2), 5 (2006)
124. Mena, E., Illarramendi, A., Royo, J.A., Gon, A.: A software retrieval service based on adaptive knowledge-driven agents for wireless environments. ACM Trans. Auton. Adapt. Syst. 1(1), 67–90 (2006)
125. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comput. Surv. 38(2), 6 (2006)
126. Chakrabarti, D., Faloutsos, C.: Graph mining: Laws, generators, and algorithms. ACM Comput. Surv. 38(1), 2 (2006)
127. Truran, M., Goulding, J., Ashman, H.: Autonomous authoring tools for hypertext. ACM Comput. Surv. 39(3), 8 (2007)
128. Jensen, E.C., Beitzel, S.M., Chowdhury, A., Frieder, O.: Repeatable evaluation of search services in dynamic environments. ACM Trans. Inf. Syst. 26(1), 1 (2007)
129. Geng, L., Hamilton, H.J.: Interestingness measures for data mining: A survey. ACM Comput. Surv. 38(3), 9 (2006)
130. Zhao, H.: Semantic matching across heterogeneous data sources. Commun. ACM 50(1), 45–50 (2007)
131. Carpenter, B.: Better, faster, more secure. Queue 4(10), 42–48 (2007)
132. Biskupski, B., Dowling, J., Sacha, J.: Properties and mechanisms of self-organizing manet and p2p systems. ACM Trans. Auton. Adapt. Syst. 2(1), 1 (2007)
133. Jelenković, P.R., Momčilović, P., Squillante, M.S.: Scalability of wireless networks. IEEE/ACM Trans. Netw. 15(2), 295–308 (2007)
134. Qiao, D., Choi, S., Shin, K.G.: Interference analysis and transmit power control in ieee 802.11a/h wireless lans. IEEE/ACM Trans. Netw. 15(5), 1007–1020 (2007)

135. Sharma, G., Mazumdar, R., Shroff, N.B.: Delay and capacity trade-offs in mobile ad hoc networks: a global perspective. IEEE/ACM Trans. Netw. 15(5), 981–992 (2007)
136. Santi, P.: Topology control in wireless ad hoc and sensor networks. ACM Comput. Surv. 37(2), 164–194 (2005)
137. Uludag, S., Lui, K.S., Nahrstedt, K., Brewster, G.: Analysis of topology aggregation techniques for qos routing. ACM Comput. Surv. 39(3), 7 (2007)
138. Androutsellis-Theotokis, S., Spinellis, D.: A survey of peer-to-peer content distribution technologies. ACM Comput. Surv. 36(4), 335–371 (2004)
139. Venugopal, S., Buyya, R., Ramamohanarao, K.: A taxonomy of data grids for distributed data sharing, management, and processing. ACM Comput. Surv. 38(1), 3 (2006)
140. Watanabe, K., Nakajima, Y., Enokido, T., Takizawa, M.: Ranking factors in peer-to-peer overlay networks. ACM Trans. Auton. Adapt. Syst. 2(3), 11 (2007)
141. Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., Guzar, A., Kartha, N., Liu, C.K., Khalaf, R., Knig, D., Marin, M., Mehta, V., Thatte, S., van der Rijn, D., Yendluri, P., Yiu, A.: Web Services Business Process Execution Language Version 2.0. Technical report, WS-BPEL TC OASIS (April 2007), http://www.oasis-open.org/
142. Misra, J., Cook, W.R.: Computation orchestration: A basis for wide-area computing. Journal of Software and Systems Modeling (to appear, 2006)
143. Cook, W.R., Patwardhan, S., Misra, J.: Workflow patterns in orc. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 82–96. Springer, Heidelberg (2006)
144. Sivasubramanian, S., Szymaniak, M., Pierre, G., van Steen, M.: Replication for web hosting systems. ACM Comput. Surv. 36(3), 291–334 (2004)
145. Saito, Y., Shapiro, M.: Optimistic replication. ACM Comput. Surv. 37(1), 42–81 (2005)
146. Elnozahy, E.N.M., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. 34(3), 375–408 (2002)
147. Bruni, R., Melgratti, H., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In: Proc. of POPL 2005, pp. 209–220. ACM Press, New York (2005)
148. Bruni, R., Melgratti, H., Montanari, U.: Composing transactional services (manuscript 2006)
149. Garcia-Molina, H., Salem, K.: Sagas. In: SIGMOD 1987: Proc. of ACM SIGMOD Int. Conf. on Management of Data, pp. 249–259. ACM Press, New York (1987)
150. Pleisch, S., Schiper, A.: Approaches to fault-tolerant and transactional mobile agent execution—an algorithmic view. ACM Comput. Surv. 36(3), 219–262 (2004)
151. Soundararajan, G., Amza, C.: Reactive provisioning of backend databases in shared dynamic content server clusters. ACM Trans. Auton. Adapt. Syst. 1(2), 151–188 (2006)
152. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. In: USENIX 2006: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation, Berkeley, CA, USA, p. 15. USENIX Association (2006)
153. Dix, A., Finlay, J., Abowd, G., Beale, R.: Human Computer Interaction, 3rd edn. Prentice-Hall, Englewood Cliffs (2004)

154. Canny, J.: The future of human-computer interaction. Queue 4(6), 24–32 (2006)
155. McTear, M.F.: Spoken dialogue technology: enabling the conversational user interface. ACM Comput. Surv. 34(1), 90–169 (2002)
156. Oviatt, S., Seneff, S.: Introduction to mobile and adaptive conversational interfaces. ACM Trans. Comput.-Hum. Interact. 11(3), 237–240 (2004)
157. Oviatt, S., Darves, C., Coulston, R.: Toward adaptive conversational interfaces: Modeling speech convergence with animated personas. ACM Trans. Comput.-Hum. Interact. 11(3), 300–328 (2004)
158. Lemon, O., Gruenstein, A.: Multithreaded context for robust conversational interfaces: Context-sensitive speech recognition and interpretation of corrective fragments. ACM Trans. Comput.-Hum. Interact. 11(3), 241–267 (2004)
159. Zhai, S., Bellotti, V.: Introduction to sensing-based interaction. ACM Trans. Comput.-Hum. Interact. 12(1), 1–2 (2005)
160. Hinckley, K., Pierce, J., Horvitz, E., Sinclair, M.: Foreground and background interaction with sensor-enhanced mobile devices. ACM Trans. Comput.-Hum. Interact. 12(1), 31–52 (2005)
161. Liao, C., Guimbretière, F., Hinckley, K., Hollan, J.: Papiercraft: A gesture-based command system for interactive paper. ACM Trans. Comput.-Hum. Interact. 14(4), 1–27 (2008)
162. Second Life: Second life web site (last accessed: 2008-06-19), http://secondlife.com/
163. Chan, H.C., Teo, H.H.: Evaluating the boundary conditions of the technology acceptance model: An exploratory investigation. ACM Trans. Comput.-Hum. Interact. 14(2), 9 (2007)
164. Bickmore, T.W., Picard, R.W.: Establishing and maintaining long-term human-computer relationships. ACM Trans. Comput.-Hum. Interact. 12(2), 293–327 (2005)
165. Crowley, J.L.: Social perception. Queue 4(6), 34–43 (2006)
166. McGrenere, J., Baecker, R.M., Booth, K.S.: A field evaluation of an adaptable two-interface design for feature-rich software. ACM Trans. Comput.-Hum. Interact. 14(1), 3 (2007)
167. Dey, A.K., Mankoff, J.: Designing mediation for context-aware applications. ACM Trans. Comput.-Hum. Interact. 12(1), 53–80 (2005)
168. Edwards, W.K.: Putting computing in context: An infrastructure to support extensible context-enhanced collaborative applications. ACM Trans. Comput.-Hum. Interact. 12(4), 446–474 (2005)
169. Sears, A., Hanson, V.L., Myers, B.: Introduction to special issue on computers and accessibility. ACM Trans. Comput.-Hum. Interact. 14(3), 11 (2007)
170. Harper, S., Bechhofer, S.: Sadie: Structural semantics for accessibility and device independence. ACM Trans. Comput.-Hum. Interact. 14(2), 10 (2007)
171. Jay, C., Glencross, M., Hubbold, R.: Modeling the effects of delayed haptic and visual feedback in a collaborative virtual environment. ACM Trans. Comput.-Hum. Interact. 14(2), 8 (2007)
172. Berstel, J., Reghizzi, S.C., Roussel, G., Pietro, P.S.: A scalable formal method for design and automatic checking of user interfaces. ACM Trans. Softw. Eng. Methodol. 14(2), 124–167 (2005)
173. Thimbleby, H.: User interface design with matrix algebra. ACM Trans. Comput.-Hum. Interact. 11(2), 181–236 (2004)
174. Sommerville, I.: Software Engineering, 8th edn. Addison-Wesley, Reading (2007)
175. Stanford Encyclopedia of Philosophy: Emergent Properties (last accessed 2008-09-21), http://plato.stanford.edu/entries/properties-emergent/

176. Holland, J.H.: Adaptation in Natural and Artificial Systems, 2nd edn. MIT Press, Cambridge (1992)
177. Bedau, M.A., Humphreys, P. (eds.): Emergence: Contemporary Readings in Philosophy and Science. MIT Press, Cambridge (2008)
178. Gros, C.: Complex and Adaptive Dynamical Systems: A Promer. Springer Complexity. Springer, Heidelberg (2008)
179. Hirsch, M.W., Smale, S., Devaney, R.L.: Differential Equations, Dynamical Systems & an Introduction to Chaos. Pure and Applied Mathematics, vol. 60. Elsevier, Amsterdam (2004)
180. Nicolis, G., Nicolis, C.: Foundations of Complex Systems: Nonlinear Dynamics, Statistical Physics, Information and Prediction. World Scientific, Singapore (2007)
181. Hirsch, D., Kramer, J., Magee, J., Uchitel, S.: Modes for software architectures. In: Gruhn, V., Oquendo, F. (eds.) EWSA 2006. LNCS, vol. 4344. Springer, Heidelberg (2006)
182. Laird, J.E.: Extending the Soar Cognitive Architecture. In: Wang, P., Goertzel, B., Franklin, S. (eds.) Proceedings of the First Artificial General Intelligence Conference. Frontiers in Artificial Intelligence and Applications, vol. 171 (February 2008)
183. Santa Fe Institute: Web Site (last accessed 2008-09-20), http://www.santafe.edu/
184. Abowd, G.D., Dey, A.K., Brown, P.J., Davies, N., Smith, M., Steggles, P.: Towards a better understanding of context and context-awareness. In: Gellersen, H.-W. (ed.) HUC 1999. LNCS, vol. 1707, pp. 304–307. Springer, Heidelberg (1999)
185. Staab, S., Studer, R. (eds.): Handbook on Ontologies. International Handbooks on Information Systems. Springer, Heidelberg (2004)
186. Turner, R.M.: Adaptive Reasoning for Real-World Problems: A Schema-Based Approach. Lawrence Erlbaum Associates, Mahwah (1994)
187. Bishop, C.M.: Pattern Recognition and Machine Learning (Information Science and Statistics). Springer, Heidelberg (2006)
188. Weber, W.: Ambient intelligence: industrial research on a visionary concept. In: ISLPED 2003: Proceedings of the 2003 international symposium on Low power electronics and design, pp. 247–251. ACM, New York (2003)
189. Babaoglu, O., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., van Moorsel, A., van Steen, M.: Self-star Properties in Complex Information Systems: Conceptual and Practical Foundations. LNCS. Springer, Heidelberg (2005)
190. Cantrill, B.: Hidden in plain sight. Queue 4(1), 26–36 (2006)
191. Patil, S., de Veciana, G.: Managing resources and quality of service in heterogeneous wireless systems exploiting opportunism. IEEE/ACM Trans. Netw. 15(5), 1046–1058 (2007)
192. Lorenz, D.H., Orda, A., Raz, D., Shavitt, Y.: Efficient qos partition and routing of unicast and multicast. IEEE/ACM Trans. Netw. 14(6), 1336–1347 (2006)
193. Movsichoff, B.A., Lagoa, C.M., Che, H.: End-to-end optimal algorithms for integrated qos, traffic engineering, and failure recovery. IEEE/ACM Trans. Netw. 15(4), 813–823 (2007)
194. Xue, G., Sen, A., Zhang, W., Tang, J., Thulasiraman, K.: Finding a path subject to many additive qos constraints. IEEE/ACM Trans. Netw. 15(1), 201–211 (2007)
195. Hirsch, D., Tuosto, E.: Coordinating application level QoS with SHReQ. Journal of Software and Systems Modelling (to appear, 2006)

196. Hirsch, D., Tuosto, E.: SHReQ: A framework for coordinating application level QoS. In: Bernhard, K., Bernhard, B. (eds.) Proceedings of SEFM 2005, 3rd IEEE International Conference on Software Engineering and Formal Methods, pp. 425–434. IEEE, Los Alamitos (2005)
197. Allman, E.: Complying with compliance. Queue 4(7), 18–21 (2006)
198. Cannon, J.C., Byers, M.: Compliance deconstructed. Queue 4(7), 30–37 (2006)
199. Buscemi, M., Montanari, U.: Cc-pi: A constraint-based language for specifying service level agreements. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 18–32. Springer, Heidelberg (2007)
200. Anderson, R.J.: Security Engineering: A Guide to Building Dependable Distributed Systems. Wiley, Chichester (2008)
201. Schneier, B.: Secrets and Lies—Digital Security in a Networked World. John Wiley & Sons, Inc., Chichester (2000)
202. Ferguson, N., Schneier, B.: Practical Cryptography. Wiley Publishing, Inc., Chichester (2003)
203. Cymru, T.: Cybercrime: an epidemic. Queue 4(9), 24–35 (2006)
204. Geer, D.E.: Playing for keeps. Queue 4(9), 42–48 (2006)
205. Kaufman, C., Perlman, R., Speciner, M.: Network security: private communication in a public world, 2nd edn. Prentice-Hall, Inc., Upper Saddle River (2002)
206. Neville-Neil, G.V.: Building secure web applications. Queue 5(5), 22–26 (2007)
207. Peng, T., Leckie, C., Ramamohanarao, K.: Survey of network-based defense mechanisms countering the dos and ddos problems. ACM Comput. Surv. 39(1), 3 (2007)
208. Kolan, P., Dantu, R.: Socio-technical defense against voice spamming. ACM Trans. Auton. Adapt. Syst. 2(1), 2 (2007)
209. Shyu, M.L., Quirino, T., Xie, Z., Chen, S.C., Chang, L.: Network intrusion detection through adaptive sub-eigenspace modeling in multiagent systems. ACM Trans. Auton. Adapt. Syst. 2(3), 9 (2007)
210. Merwe, J.V.D., Dawoud, D., McDonald, S.: A survey on peer-to-peer key management for mobile ad hoc networks. ACM Comput. Surv. 39(1), 1 (2007)
211. Rafaeli, S., Hutchison, D.: A survey of key management for secure group communication. ACM Comput. Surv. 35(3), 309–329 (2003)
212. Tolone, W., Ahn, G.J., Pai, T., Hong, S.P.: Access control in collaborative systems. ACM Comput. Surv. 37(1), 29–41 (2005)
213. Zhang, G., Baumeister, H., Koch, N., Knapp, A.: Aspect-Oriented Modeling of Access Control in Web Applications. In: Proc. 6th Int. Wsh. Aspect Oriented Modeling (WAOM 2005), Chicago (2005)
214. Bhatti, R., Bertino, E., Ghafoor, A.: An integrated approach to federated identity and privilege management in open systems. Commun. ACM 50(2), 81–87 (2007)
215. Geer, D.E.: The evolution of security. Queue 5(3), 30–35 (2007)
216. Boyle, M., Greenberg, S.: The language of privacy: Learning from video media space analysis and design. ACM Trans. Comput.-Hum. Interact. 12(2), 328–370 (2005)
217. Loscocco, P., Smalley, S.: Integrating flexible support for security policies into the linux operating system (February 2001) (last accessed 2008-01-19), http://www.nsa.gov/selinux/papers/slinux-abs.cfm
218. Drepper, U.: Security enhancements in red hat enterprise linux (beside selinux) (December 2005) (last accessed 2008-01-19), http://people.redhat.com/drepper/nonselsec.pdf
219. Ford, R.: Open vs. closed: which source is more secure? Queue 5(1), 32–38 (2007)
220. Hoepman, J.H., Jacobs, B.: Increased security through open source. Commun. ACM 50(1), 79–83 (2007)

221. Thayer, R.H.: Software system engineering: A tutorial. IEEE Computer 35(4), 68–73 (2002)
222. White, S., Alford, M.W., Holtzman, J., Kuehl, C.S., McCay, B., Oliver, D., Owens, D., Tully, C., Willey, A.: Systems engineering of computer-based systems, state of practice working group. IEEE Computer 26(11), 54–65 (1993)
223. IEEE: Std. 1220-1998. Standard for Application and Management of the System Engineering Process. IEEE Press, Piscataway, N.J (1998)
224. George, W., Beeler, J., Gardner, D.: A requirements primer. Queue 4(7), 22–26 (2006)
225. Giorgini, P., Massacci, F., Zannone, N.: Security and Trust Requirements Engineering. In: Aldini, A., Gorrieri, R., Martinelli, F. (eds.) FOSAD 2005, vol. 3655, pp. 237–272. Springer, Heidelberg (2005)
226. Donzelli, P., Basili, V.R.: A practical framework for eliciting and modeling system dependability requirements: Experience from the nasa high dependability computing project. Journal of Systems and Software 79(1), 107–119 (2006)
227. Cantor, M., Roose, G.: Hardware/software codevelopment using a model-driven systems development (mdsd) approach. IBM developerWorks (December 2005) (last accessed 2008-01-20), http://www.ibm.com/developerworks/rational/library/dec05/cantor/
228. Cantor, M., Roose, G.: Hardware/software codevelopment using a model-driven systems development (mdsd) approach—part ii: Illustrating the solution. IBM developerWorks (Febuary 2006) (last accessed 2008-01-20), http://www.ibm.com/developerworks/rational/library/feb06/cantor-roose/
229. Unrau, R.C.: Development techniques for using simulation to remove risk in software/hardware integration (2008) (last accessed 2008-01-20), http://www.redhat.com/support/wpapers/cygnus/cygnus_risk/index.html#toc
230. Robinson, W.N., Pawlowski, S.D., Volkov, V.: Requirements interaction management. ACM Comput. Surv. 35(2), 132–190 (2003)
231. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide, 2nd edn. Addison-Wesley, Reading (2005)
232. Mellor, S.J., Kendall, S., Uhl, A., Weise, D.: MDA Distilled. Addison Wesley Longman Publishing Co., Inc., Redwood City (2004)
233. Object Management Group: Model Driven Architecture (OMG) (Last visited, June 2008), http://www.omg.org/mda/
234. Paige, R.F., Brooke, P.J., Ostroff, J.S.: Metamodel-based model conformance and multiview consistency checking. ACM Trans. Softw. Eng. Methodol. 16(3), 11 (2007)
235. Beck, K.: Extreme Programming Explained—Embracing Change. Addison-Wesley Professional, Reading (2003)
236. Cockburn, A.: Agile Software Development—The Cooperative Game, 2nd edn. Addison-Wesley, Reading (2006)
237. Highsmith, J.: Agile Software Development Ecosystems. Addison-Wesley, Reading (2003)
238. Nerur, S., Balijepally, V.: Theoretical reflections on agile development methodologies. Commun. ACM 50(3), 79–83 (2007)
239. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. The SEI Series in Software Engineering. Addison-Wesley Professional, Reading (2001)
240. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools and Applications. Addison-Wesley Professional, Reading (2000)
241. Greenfield, J., Short, K.: Software Factories: Assembling Applications with Patterns, Frameworks, Models and Tools. Wiley, Chichester (2004)

242. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, Reading (2004)
243. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman, Amsterdam (1995)
244. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Longman, Amsterdam (2002)
245. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture. A System of Patterns, vol. 1. John Wiley & Sons, Chichester (1996)
246. Schmidt, D., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture. Patterns for Concurrent and Networked Objects, vol. 2. John Wiley & Sons, Chichester (2000)
247. Kircher, M., Jain, P.: Pattern-Oriented Software Architecture. Patterns for Resource Management, vol. 3. Wiley, Chichester (2004)
248. Buschmann, F., Henney, K., Schmidt, D.C.: Pattern-Oriented Software Architecture. A Pattern Language for Distributed Computing, vol. 4. Wiley, Chichester (2007)
249. Buschmann, F., Henney, K., Schmidt, D.C.: Pattern Oriented Software Architecture. On Patterns and Pattern Languages, vol. 5. Wiley, Chichester (2007)
250. Gay, D., Levis, P., Culler, D.: Software design patterns for tinyos. Trans. on Embedded Computing Sys. 6(4), 22 (2007)
251. Binder, R.: Testing Object-Oriented Systems: Models, Patterns and Tools. Addison-Wesley Professional, Reading (2000)
252. Rothermel, G., Elbaum, S., Malishevsky, A.G., Kallakuri, P., Qiu, X.: On test suite composition and cost-effective regression testing. ACM Trans. Softw. Eng. Methodol. 13(3), 277–331 (2004)
253. Kaufmann, M., Manolis, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Dordrecht (2000)
254. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS (LNAI), vol. 607, pp. 748–752. Springer, Heidelberg (1992)
255. McDonald, J., Anton, J.: Specware - producing software correct by construction (2001)
256. Bertot, Y., Castéran, P., Huet, G., Paulin-Mohring, C.: Interactive Theorem Proving and Program Development. Springer, Heidelberg (2004)
257. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer, Heidelberg (2002)
258. Russinoff, D., Kaufmann, M., Smith, E., Summers, R.: Formal verification of floating-point rtl at amd using the acl2 theorem prover. In: Simonov, N. (ed.) Proceedings of the 17th IMACS World Congress on Scientific Computing (July 2005)
259. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley Professional, Reading (2004)
260. Spin: Web site of the SPIN Model Checker (2008) (last visited: 2008-01-24), www.spinroot.com
261. Edmund, M., Clarke, O.G., Peled, D.A.: Model Checking. MIT Press, Cambridge (2000)
262. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Software verification with blast. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003, vol. 2648, pp. 235–239. Springer, Heidelberg (2003)

263. Fraser, G., Wotawa, F.: Improving model-checkers for software testing. qsic 0, 25–31 (2007)
264. Fraser, G., Wotawa, F.: Using ltl rewriting to improve the performance of model-checker based test-case generation. In: A-MOST, pp. 64–74. ACM, New York (2007)
265. Fraser, G., Wotawa, F.: Redundancy based test-suite reduction. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 291–305. Springer, Heidelberg (2007)
266. Ren, S., Yu, Y., Chen, N., Tsai, J.J.P., Kwiat, K.: The role of roles in supporting reconfigurability and fault localizations for open distributed and embedded systems. ACM Trans. Auton. Adapt. Syst. 2(3), 10 (2007)
267. Papadopoulos, P., Bruno, G., Katz, M.: Beyond beowulf clusters. Queue 5(3), 36–43 (2007)
268. Erl, T.: Service-Oriented Architecture—Concepts, Technology and Design. Prentice Hall Service-Oriented Computing Series. Prentice-Hall, Englewood Cliffs (2005)
269. Edwards, D., Simmons, S., Wilde, N.: An approach to feature location in distributed systems. Journal of Systems and Software 79(1), 57–68 (2006)
270. Babaoglu, O., Canright, G., Deutsch, A., Caro, G.A.D., Ducatelle, F., Gambardella, L.M., Ganguly, N., Jelasity, M., Montemanni, R., Montresor, A., Urnes, T.: Design patterns from biology for distributed computing. ACM Trans. Auton. Adapt. Syst. 1(1), 26–66 (2006)
271. Banâtre, J.P., Fradet, P., Radenac, Y.: A generalized higher-order chemical computation model. Electr. Notes Theor. Comput. Sci. 135(3), 3–13 (2006)
272. Agha, G.: Computing in pervasive cyberspace. Commun. ACM 51(1), 68–70 (2008)
273. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge (1986)
274. Agha, G.A., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. J. Funct. Program. 7(1), 1–72 (1997)
275. Wooldridge, M.: An introduction to multi-agent systems. J. Artificial Societies and Social Simulation 7(3) (2004)
276. Durfee, E.H., Yokoo, M., Huhns, M.N., Shehory, O. (eds.): 6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2007), Honolulu, Hawaii, USA, IFAAMAS, May 14-18 (2007)
277. Talcott, C.L.: A formal framework for interactive agents. Electron. Notes Theor. Comput. Sci. 203(3), 95–106 (2008)
278. Poslad, S.: Specifying protocols for multi-agent systems interaction. ACM Trans. Auton. Adapt. Syst. 2(4), 15 (2007)
279. Penserini, L., Perini, A., Susi, A., Mylopoulos, J.: High variability design for software agents: Extending tropos. ACM Trans. Auton. Adapt. Syst. 2(4), 16 (2007)
280. Giorgini, P., Mouratidis, H., Zannone, N.: Modelling Security and Trust with Secure Tropos. In: Mouratidis, H., Giorgini, P. (eds.) Integrating Security and Software Engineering: Advances and Future Vision. Idea Group (2007)
281. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services: Concepts, Architectures and Applications. Springer, Heidelberg (2004)
282. Paurobally, S., Tamma, V., Wooldrdige, M.: A framework for web service negotiation. ACM Trans. Auton. Adapt. Syst. 2(4), 14 (2007)

283. Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: Choreography and Orchestration: a synergic approach for system design. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005, vol. 3826, pp. 228–240. Springer, Heidelberg (2005)
284. Foster, H., Uchitel, S., Kramer, J., Magee, J.: Ws-engineer: A tool for model-based verification of web service compositions and choreography. In: IEEE International Conference on Software Engineering (ICSE 2006), Shanghai, China (May 2006)
285. van Breugel, F., Koshkina, M.: Models and verification of BPEL (2006), http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf
286. Project, N.: Web-site of the NESSI technology platform (2007), www.nessi-europe.com
287. BIONETS project: Web site of the BIONETS project (last accessed: 2008-06-19), http://www.bionets.eu/
288. CASCADAS project: Web site of the CASCADAS project (last accessed: 2008-06-19), http://www.cascadas-project.org/
289. ONTOGRID project: Web site of the ONTOGRID project (last accessed: 2008-06-19), http://www.ontogrid.net/
290. SIMS project: Web site of the SIMS project (last accessed: 2008-06-19), http://www.ist-sims.org/
291. SODIUM project: Web site of the SODIUM project (last accessed: 2008-06-19), http://www.atc.gr/sodium/index.asp
292. PLASTIC project: Web site of the PLASTIC project (last accessed: 2008-06-19), http://www.ist-plastic.org/
293. SENSORIA: Software Engineering for Service-Oriented Overlay Computers, http://www.sensoria-ist.eu
294. ONE project: Web site of the ONE project (last accessed: 2008-06-19), http://one-project.eu/site/modules/content/index.php?id=1
295. ASTRO project: Web site of the ASTRO project (last accessed: 2008-06-19), http://www.astroproject.org/
296. AOSD project: Web site of the AOSD project (last accessed: 2008-06-19), http://www.aosd-europe.net/
297. MUSIC project: Web site of the MUSIC project (last accessed: 2008-06-19), http://www.ist-music.eu/
298. WSDIAMOND project: Web site of the WS-DIAMOND project (last accessed: 2008-06-19), http://wsdiamond.di.unito.it/
299. SECSE project: Web site of the SECSE project (last accessed: 2008-06-19), http://secse.eng.it/
300. INFRAWEBS project: Web site of the INFRAWEBS project (last accessed: 2008-06-19), http://www.infrawebs.org/
301. DIP project: Web site of the DIP project (last accessed: 2008-06-19), http://dip.semanticweb.org/
302. AMIGO project: Web site of the AMIGO project (last accessed: 2008-06-19), http://www.hitech-projects.com/euprojects/amigo/
303. WS2 project: Web site of the WS2 project (last accessed: 2008-06-19), http://www.w3.org/2004/WS2/
304. ESFORS project: Web site of the ESFORS project (last accessed: 2008-06-19), http://www.esfors.org/
305. S3MS project: Web site of the S3MS project (last accessed: 2008-06-19), http://www.s3ms.org/
306. TRUSTCOM project: Web site of the TRUSTCOM project (last accessed: 2008-06-19), http://www.eu-trustcom.com/

307. ATHENA project: Web site of the ATHENA project, http://www.athena-ip.org/
308. DEDISYS project: Web site of the DEDISYS project (last accessed: 2008-06-19), http://www.dedisys.org/
309. di Nitto, E., Traverso, P., Sassen, A.M., Zwegers, A. (eds.): At your service: An overview of results of projects in the field of service engineering of the IST programme. MIT Press, Cambridge (2008)
310. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
311. Elrad, T., Filman, R.E., Bader, A.: Aspect-oriented programming: Introduction. Commun. ACM 44(10), 29–32 (2001)
312. Elrad, T., Aksit, M., Kiczales, G., Lieberherr, K., Ossher, H.: Discussing aspects of aop. Commun. ACM 44(10), 33–38 (2001)
313. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
314. Lieberherr, K., Orleans, D., Ovlinger, J.: Aspect-oriented programming with adaptive methods. Commun. ACM 44(10), 39–41 (2001)
315. Bergmans, L., Aksit, M.: Composing crosscutting concerns using composition filters. Commun. ACM 44(10), 51–57 (2001)
316. Aksit, M., Bergmans, L., Vural, S.: An object-oriented language-database integration model: The composition-filters approach. In: Lehrmann Madsen, O. (ed.) ECOOP 1992, vol. 615, pp. 372–395. Springer, Heidelberg (1992)
317. Ossher, H., Tarr, P.: Using multidimensional separation of concerns to (re)shape evolving software. Commun. ACM 44(10), 43–50 (2001)
318. Tarr, P., Ossher, H., Harrison, W., Stanley, M., Sutton, J.: N degrees of separation: multi-dimensional separation of concerns. In: ICSE 1999: Proceedings of the 21st international conference on Software engineering, pp. 107–119. IEEE Computer Society Press, Los Alamitos (1999)
319. Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.): MODELS 2007. LNCS, vol. 4735. Springer, Heidelberg (2007)
320. Smith, B.C.: Reflection and semantics in lisp. In: POPL 1984: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 23–35. ACM, New York (1984)
321. Maes, P.: Concepts and experiments in computational reflection. SIGPLAN Not. 22(12), 147–155 (1987)
322. Kiczales, G., des Rivières, J., Bobrow, D.G.: The Art of the Metaobject Protocol. MIT Press, Cambridge (1991)
323. Sullivan, G.T.: Aspect-oriented programming using reflection and metaobject protocols. Commun. ACM 44(10), 95–97 (2001)
324. Sullivan, G.T.: Dynamic partial evaluation. In: Danvy, O., Filinski, A. (eds.) PADO 2001, vol. 2053, pp. 238–256. Springer, Heidelberg (2001)
325. Costanza, P., Hirschfeld, R.: Reflective layer activation in contextl. In: SAC 2007, pp. 1280–1285. ACM, New York (2007)
326. Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming: an overview of contextl. In: DLS 2005: Proceedings of the 2005 symposium on Dynamic languages, pp. 1–10. ACM, New York (2005)
327. Krishnamurthi, S., Fisler, K.: Foundations of incremental aspect model-checking. ACM Trans. Softw. Eng. Methodol. 16(2), 7 (2007)

328. Misra, A., Karsai, G., Sztipanovits, J.: Model-integrated development of complex applications. In: SAST 1997: Proceedings of the 5th International Symposium on Assessment of Software Tools (SAST 1997), Washington, DC, USA, p. 14. IEEE Computer Society Press, Los Alamitos (1997)
329. Karsai, G., Ledeczi, A., Neema, S., Sztipanovits, J.: The model-integrated computing toolsuite: Metaprogrammable tools for embedded control system design. In: Proc. of the IEEE Joint Conference CCA, ISIC and CACSD, Munich, Germany, pp. 50–55 (October 2006)
330. Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., Goedicke, M.: Viewpoints: a framework for integrating multiple perspectives in system development. International Journal of Software Engineering and Knowledge Engineering 2(1), 31–57 (1992)
331. Finkelstein, A.: Relating viewpoints. In: ACM SIGSOFT 1996 Workshop - Viewpoints 1996, p. 157. ACM Press, New York (1996)
332. Tekinerdogan, B., Hofmann, C., Aksit, M.: Modeling traceability of concerns in architectural views. In: AOM 2007: Proceedings of the 10th international workshop on Aspect-oriented modeling, pp. 49–56. ACM, New York (2007)
333. Odersky, M., Spoon, L., Venners, B.: Programming in Scala: A comprehensive step-by-step guide. PrePrint Edition Version 4. Artima Developer (August 2008), http://www.artima.com/shop/programming_in_scala
334. Research, M.: F#: A succinct, type-inferred, expressive, efficient functional and object-oriented language for the .net platform (2008) (last accessed: 2008-09-22), http://research.microsoft.com/fsharp/
335. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. 37(4), 316–344 (2005)
336. Robillard, M.P., Murphy, G.C.: Representing concerns in source code. ACM Trans. Softw. Eng. Methodol. 16(1), 3 (2007)
337. Bacon, D.F.: Realtime garbage collection. Queue 5(1), 40–49 (2007)
338. Ashley-Rollman, M.P., De Rosa, M., Srinivasa, S.S., Pillai, P., Goldstein, S.C., Campbell, J.D.: Declarative programming for modular robots. In: Workshop on Self-Reconfigurable Robots/Systems and Applications at IROS 2007 (October 2007)
339. Ashley-Rollman, M.P., Goldstein, S.C., Lee, P., Mowry, T.C., Pillai, P.: Meld: A declarative approach to programming ensembles. In: Proceedings of the IEEE International Conference on Intelligent Robots and Systems IROS 2007 (October 2007)
340. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang, 2nd edn. Prentice Hall, Englewood Cliffs (1996)

Software Engineering for Ensembles*

Matthias Hölzl, Axel Rauschmayer, and Martin Wirsing

Ludwig-Maximilians-Universität München

Abstract. Software development is difficult, even if we control most of the operational parameters and if the software is designed to run on a single machine. But in the future we will face an even more challenging task: engineering ensembles consisting of thousands, or even millions, of nodes, all operating in parallel, with open boundaries, possibly unreliable components and network links, and governed by multiple entities. To develop reliable and trustworthy software for these kinds of systems we need to go far beyond the current state of the art and address fundamental problems in software development. We present some challenges and promising avenues for research about software-engineering for ensembles.

1 Introduction

"I found that writing software was much more difficult than anything else I had done in my life." These words from well-known computer scientist Donald E. Knuth [1] illustrate the challenges faced by software engineers, even when writing traditional software. And yet the situation is about to become even more demanding: we are moving from isolated applications running in a fairly well-determined environment to *ensembles*—software-intensive systems with massive numbers of nodes, operating in open and non-deterministic environments in which they have to interact with humans or other software-intensive systems. Ensembles will have to dynamically adapt to new requirements, technologies or environmental conditions without redeployment and without interruption of the system's functionality, thereby blurring the distinction between design-time and run-time.

This move from engineering traditional systems to ensembles is not triggered by idle desire on the part of software engineers; large, networked software-intensive systems are already forming indispensable parts of our infrastructure, from power grids to financial trading systems, and failures of these systems can have dramatic consequences for our economy and well-being.

While many of the problems posed by traditional software are difficult but manageable, some of the challenges posed by ensembles are beyond the current state of the art for software and systems engineering. For example, while some methods exist for building systems with a limited amount of adaptivity in restricted contexts, we are not aware of any software system that can reliably

* This work has been partially sponsored by the projects SENSORIA, IST-2 005-016004 and InterLink, IST-FET CN 034051.

M. Wirsing et al. (Eds.): Software-Intensive Systems, LNCS 5380, pp. 45–63, 2008.

adapt to changing requirements in realistic, open-ended scenarios over significant time frames; and it is moreover not yet clear which theoretical foundations, engineering principles and development tools are really adequate for building such a system.

That is not to say that we cannot build complex systems; the number of systems currently in operation shows that we have some successes when building systems with current engineering techniques. However, these systems are generally difficult and expensive to build and maintain, often brittle in the face of unexpected circumstances, and too frequently plagued by security holes and reliability problems.

Therefore we need a two-pronged approach to develop a discipline of software-engineering for ensembles. We need incremental improvements to the current practice for developing systems, and we need to investigate radical and transformative ideas that may push the subject in new directions. In the first strand of research it will be easier to achieve results that can be integrated into traditional software development processes, but it is unlikely that we will achieve the above-stated goals for ensembles without fundamental changes in the way we understand and build systems.

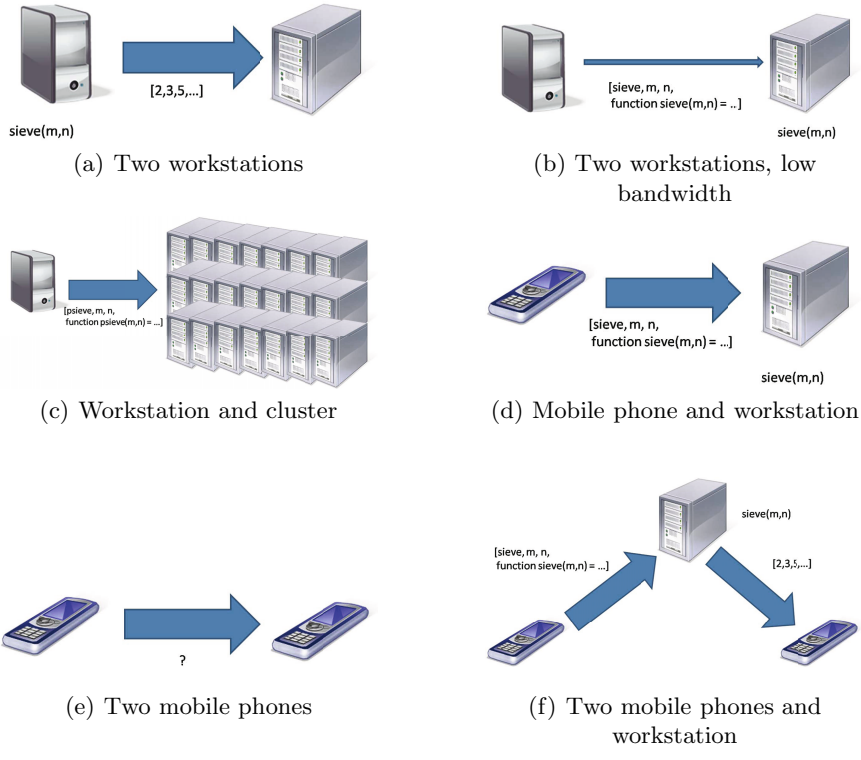
In the rest of the paper we will address some of the problems posed by ensembles and propose some directions for future research—both incremental and disruptive—that we consider to be promising.

2 Difficulties in Software-Development

In his influential articles [2,3] Frederick P. Brooks pointed out that the difficulty in the development of software has both *essential* and *accidental* factors. To paraphrase, essential difficulties are those inherent in understanding and addressing the problem the software system is meant to solve, while accidental difficulties are those arising from the tools and methods used in the production of the software. In [2] Brooks identified four main reasons for essential difficulties in the development of software: the *complexity* of the problems that software addresses, the requirement of *conformity* to complex external interfaces and business processes, the *changeability* of software which leads to continuous demands for additional functionality, and the *invisibility* of software artifacts which prevents us from using physical or spatial intuitions. Brooks also suggested four promising ways to address these difficulties which might be summarized as:

- Buying software or components instead of building from scratch.
- Iterative development and rapid prototyping instead of waterfall models.
- Growing software organically, i.e., gradually adding features to a system instead of trying to specify and build the complete system from the outset.
- Recruiting and training great conceptual designers.

While there have been some arguments about the details of Brooks's theses, it is interesting to note that the essence of both papers is still relevant today, more than 20 years after the first paper was written; the principles advocated by supporters of agile development methods seem to echo many of these results.



(a) Two workstations

(b) Two workstations, low bandwidth

(c) Workstation and cluster

(d) Mobile phone and workstation

(e) Two mobile phones

(f) Two mobile phones and workstation

Fig. 1. Possible environments for the prime computation

Developing software for ensembles creates new difficulties related to the scale of the systems, the need to adapt to changing environments and requirements, emergent properties resulting from interactions between nodes, and uncertainty during design-time and run-time. These difficulties are not accidental and probably will not succumb to "silver bullet" solutions.

To illustrate some of the challenges, we consider a very simple, if artificial, problem: *communicate all prime number in the interval* $[m, n]$ *from node* A *to node* B. There are various well-known algorithms for solving this problem [4], e.g., node A might use the Sieve of Eratosthenes or a probabilistic primality test to compute all primes in the range $[m, n]$ and then transmit this list to node B.

In the description of the solution we have made several implicit assumptions that may not be correct when we develop for ensembles: we have assumed that the network capacity between the nodes is sufficient to transmit the list of primes and that node A is sufficiently powerful to compute the solution. In Figure 1 we have depicted different scenarios that may occur in an ensemble: Fig. 1(a) represents the original situation: two workstations connected by a high-bandwidth network. In Fig. 1(b) two workstations are connected by a low-bandwidth connection. In this case it may not be feasible to transmit a long list of primes to node B, instead it might be necessary to transmit the individual results when

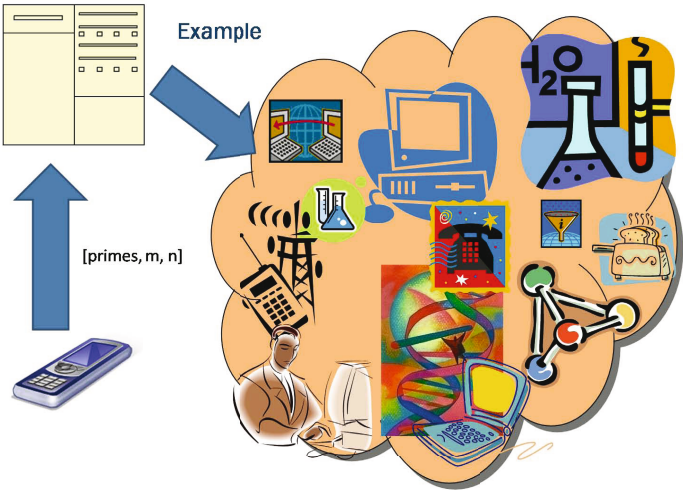


Fig. 2. Probable environment in an ensemble

they are needed, or even a program to compute the primes. In Fig. 1(c) node B represents a powerful cluster where it would be advantageous to use a parallel algorithm. It might also happen, as depicted in Fig. 1(d) that the program is running on a mobile phone that does not itself possess the resources to compute the desired solution. In that case an appropriate solution would again be to transmit the program to compute the primes to node B. However, if the situation is as in Fig. 1(e) and two devices with limited resources are communicating, then this solution is no longer applicable. Instead the devices might try to discover another node C that can perform the computation on their behalf and transmit the results to B.

In order to successfully operate in an ensemble, even our simple program would have to acquire significant complexity, with different strategies to solve the problem, depending on intricate knowledge of its execution platform and its environment. To keep the example manageable we did not try to take into account many other complications arising in real systems, e.g., the scale or nondeterminism of the system, the need to minimize communication costs, fulfilling service-level agreements, or security considerations. Furthermore, in future ensembles the situation will rarely be as simple as in Fig. 1, a more likely scenario is Fig. 2, where the environment contains human operators, physical devices, nodes operating according to new computing paradigms, etc. It is evident that manually specifying how to deal with all possible interactions will not be feasible except for the simplest ensembles. Instead we need formalisms and languages that allow developers to focus on individual aspects at different stages in the development process and then combine the solutions into a coherent program.

In the next section we will argue that we consider model-based approaches mandatory for a successful discipline of ensemble engineering, but that "model-based development" as it is currently understood in the software-engineering

community has some deficiencies that need to be rectified to fully support the development of ensembles. The subsequent sections on adaptation and emergence, distributed development, development tools, and domain-specific development, all expand on the idea of model-based development.

3 Modeling and Programming for Ensembles

The most successful strategy to combat complexity is to address the problem at a higher level of abstraction. This trend has been visible throughout the history of computing, where we have moved from assembly language to modern high-level programming and modeling languages. Currently, object-oriented techniques and languages are prevalent in the development of most new software systems; UML models [5] are routinely used, and have mostly replaced other modeling notations in mainstream software development. While these techniques have been successful for traditional software development, there are several issues with current modeling approaches when considering the requirements of ensembles:

- The formalisms used to model systems do not have a well-defined semantics.
- The executable code has no way to reference its model.
- Models are mostly "shallow."

3.1 Semantics of Models

There are many types of models defined in the UML specification, but the semantics of UML is often complex or non-intuitive, and sometimes inconsistent: the UML specification contains many extension points which deliberately leave certain aspects of the language underspecified. An example of a semantic problem in the definition of UML 2.0 is given in [6], where the authors argue that the definition of associations has semantic implications which are not represented in the syntax and may easily be misunderstood. The article [7] by Steve Cook and Stuart Kent contains a a more detailed discussion of problems with the current UML specification, in particular for the generation of executable code from UML models and the definition of domain-specific extensions for UML. We will argue in the next subsection that the model should be available for inspection and reasoning at run-time; a simple, well-defined semantics is therefore mandatory.

On the other hand, UML is often useful to guide discussions between stakeholders and developers, while typical knowledge-representation or specification languages require a degree of mathematical sophistication from their users that cannot be expected in typical development projects.

An important challenge is therefore to develop modeling languages that combine simple, well-specified semantics with high expressivity and good usability for developers. Those languages used to communicate with stakeholders should also be understandable for domain experts and customers. When a suitable language is already widely used in a certain domain, mapping it into the modeling language should be semantically straightforward and achievable by automated tools. This

is related to research in the areas of model-integrated development [8,9] and model-driven architecture [10,11], where domain models are used to generate executable code and have therefore to be equipped with a precise semantics.

3.2 Robust Formal Methods

A semantically well-defined modeling language enables the use of formal methods, such as model checking or theorem proving, during design time and at run time. While formal methods are not yet widely used in industrial software development, considerable progress has been made in developing efficient, automated tools that can be used by developers to increase the quality of software. The book by Johann M. Schumann [12] contains an overview of the state of the art in the area, as of 2001; more recent advances can be found, e.g., in the proceedings of CADE [13] or IJCAR [14]. Improvements in implementation technology and theoretical discoveries have led to tools that can solve increasingly large problems automatically, e.g., John Rushby argues in [15] that SMT (satisfiability modulo theories) solvers are an example for this sort of disruptive innovation.

However, one of the problems with formal methods is that their behavior is often unpredictable: only the most expert users can quickly and accurately estimate whether a certain model or a certain set of parameters can be checked given the available resources. If formal methods are to play an important part in the run-time environment of ensembles they will have to be more robust, i.e., the run-time has to be able to reliably determine which tools are applicable in a given situation, e.g., by being able to estimate the expected execution time and the quality of the expected results for various tools.

3.3 Connecting the Code to Its Models

In current software development practice models are used to develop or generate the code, but the code has no possibility to access its models during runtime. Being able to reason about their models and environments would open many avenues for programs to adapt to unforeseen circumstances: to cope with the situations presented in Fig. 1 it is not enough for a program to have a subroutine that computes a list of primes, it needs to have a model of its environment and connectivity, and it needs to know the resources needed by various methods to check for primality in order to choose an appropriate strategy. It would be even better if the different nodes in the systems had a shared notion of "checking primality of a number" so that each node could choose an appropriate strategy according to its capabilities.

To achieve this, code, models, and requirements have to be closely integrated: the program has to be able to determine, while it is executing, which models are relevant to the executing code; on the other hand, if new requirements cause a model to be changed, the code implementing this model has to be modified on the fly to satisfy the updated specifications. Irrespective of automated adaptation of programs taking place, this kind of *traceability* [16,17] between requirements, models, and code, offers great advantages for the development and debugging of programs.

As for reasoning about models, several powerful specification and verification systems are currently available and used in industrial applications, e.g., ACL2 [18], PVS [19,20], or Kestrel Specware [21]. These would be prime candidates for creating program-integrated models, since their ability to create useful specification that can, in many cases, directly be executed has already been established. However they are targeted at human developers and cannot directly be used to reason about programs in a fully automated manner:

- Provers for these systems are interactive and rely on the developer to provide required lemmata or to discharge proof obligations. Integrating these systems into programs would require fully automated theorem provers, which are currently not feasible for their powerful logics; it would require a breakthrough in the development of theorem prover technology to develop a system that can decide a large number of interesting problems without human intervention.
- Proofs of interesting properties often take a long time, even with human guidance. This time may not be available when the prover is used to reconfigure a software component.
- The provers often use more resources than typical nodes in an ensemble may have.

There are two research directions that promise significant long-term progress: one is the development of formalisms that are expressive enough to model interesting program properties while having low computational complexity, at least for a practically relevant class of problems. Interesting developments on this front are currently taking place, e.g., in the area of description logics [22] and temporal logics [23]. Modal logics [24], which allow the reasoning about the belief of other agents and may be appropriate for programs that try to reason in depth about their environment. Interdisciplinary research with areas such as knowledge representation [25,26,27], expert systems [28], or model-based diagnostics [29] may be effective.

Another promising research direction is the integration of approaches such as theory resolution [30] or special-purpose decision-procedures into general-purpose reasoning systems [31]. While the research in this area has a long history, promising results have recently been obtained, e.g., by the Cyc project [32]: the study [33] shows that the Cyc reasoner, which contains a large number of special-purpose decision procedures for the huge Cyc knowledge base of common-sense knowledge, outperforms a sophisticated first-order prover by several orders of magnitude on a number of decision problems. We return to this problem in Section 7.

3.4 Surface and Deep Models

The distinction between "surface systems" and "deep systems" was defined in an article by Peter E. Hart [34] as follows:

> By surface systems I mean those having no underlying representation of such fundamental concepts as causality, intent, or basic physical principles; deep systems, by contrast, attempt to represent concepts at this level.

While the distinction between deep systems (also called "causal models" [35]) and surface systems (which have also been called "shallow models," [29] "empirical associations," [35] or "compiled-knowledge systems" [36] in the literature) is neither objective nor unambiguous, it nevertheless expresses an important difference. Most models that we use in software engineering can be classified as surface models, since they express the information required for the software to function but not the reasons why the system behaves the way it does, or what consequences the actions of the system have.

For example, a model of a university management system typically includes associations between the student and university or course, but it does not contain enough information to conclude that being exmatriculated may have serious consequences for a student, or that posting exam questions on the web may not be advisable before the exam takes place (but might be permissible after the exam is over). This is not problematic when the models are used by developers as an aid in the development process, or when they are used for code generation purposes, and therefore the greater simplicity and lower cost to build surface models is justified. However, when models guide (non-interactive) adaptation, inclusion of "deep" knowledge in models seems to be almost mandatory. How this can be achieved in practice is still largely a question for future research; the free availability of the OpenCyc knowledge base [37] presents many interesting opportunities for research in this area. There may also be interesting confluences with research on rationale management [38] which tries to capture the design rationales of software developers. Interesting research direction in this area may also include the combination of structural causality models [39,40] with models that represent the effects of actions, such as the situation calculus [41,42], the event calculus [43]. The article [44] represents a first step in this direction.

4 Adaptation and Emergence

We call *adaptation* the capability of a system to change its behavior according to new requirements or environment conditions. The term *emergence* has been used to describe various phenomena: in the software engineering literature it is often used to describe global phenomena, not arising from any single component [45]; in the literature about complex system it is often used with more specific denotations, for example Mark A. Bedau defines *weak emergence* as [46]: "Macrostate P of [a system] S with microdynamic D is weakly emergent iff P can be derived from D and S's external conditions but only by simulation." In this section it is mostly this latter denotation of emergence that we are concerned with.

4.1 Adaptation Mechanisms

To formalize the discussion of adaptation we assume that we have models of the program (*Prog*), the environment (*Env*), the connection between the program and the environment (*Link*) and the desired result of the computation (*Res*). These models can be expressed at various levels of abstraction, depending on

the details of the situation. For example, *Prog* might be a non-executable specification of the program, or it might be the executable program. We assume, however, that we have a semantic consequence relation $\models$ available. Then we can describe the correctness of the program in its environment as follows:

$$Prog, Env, Link \models Res. \tag{1}$$

A weaker[1], but often useful, formulation is to ensure that the correct result is not incompatible with the program in a given environment, although the correct result need not be implied (i.e., *Prog*, *Env*, *Link* and *Res* are consistent):

$$Prog, Env, Link, Res \not\models \bot. \tag{2}$$

This latter version allows, e.g., reasoning in many cases where some of the data in the models is still unknown. Note that these formalizations are very similar to those used in model-based problem solving [29].

If we call $Prog_1$ the program that computes the primes between m and n on node A, $Prog_2$ the program that transmits the code to node B, and $Env(i)$ ($Link(i)$) the environment (links) depicted in Fig. i, then we have:

$$Prog_1, Env(1(a)), Link(1(a)) \models Res \qquad Prog_2, Env(1(a)), Link(1(a)) \models Res$$
$$Prog_1, Env(1(d)), Link(1(d)) \not\models Res \qquad Prog_2, Env(1(d)), Link(1(d)) \models Res$$
$$Prog_1, Env(1(e)), Link(1(e)) \not\models Res \qquad Prog_2, Env(1(e)), Link(1(e)) \not\models Res$$

The first two equations state that either computing the result on node A or transmitting the program to node B leads to a correct result in the case of two connected workstations, the third equation states that the mobile phone is not able to compute the solution on its own while the fourth equation states that it can compute the solution by transmitting the program to the workstation. The last two equations demonstrate that two connected phones have no way to compute the solution (without reconfiguration), no matter which of the two programs they use.

Adaptation of a program to a changing environment denotes the following process: we have a program that works correctly in a certain environment and configuration

$$Prog, Env, Link \models Res$$

and the environment changes to Env'. We are now looking for a new program $Prog'$ such that[2]

$$Prog', Env', Link \models Res.$$

Adaptation to changed requirements is, in the presented formalism, similar to adaptation to a changed environment. In this case, instead of modifying the

[1] We assume that $Prog, Env, Link \not\models \bot$.

[2] If the result *Res* depends on the environment, then instead of *Res* the adapted program $Prog'$ may have to satisfy a different result Res' which can be derived from *Res* and the changes in the environment.

environment model, the result model *Res* is replaced by the updated model *Res*′. Similarly, reconfiguration can be modeled by modifying *Link* instead of *Prog*.

The example in Fig. 1 illustrates the simplest case for adaptation: we have a fixed number of programs ($Prog_1$, $Prog_2$) that we can substitute for each other. This essentially corresponds to uses of the *Strategy* pattern. A more interesting example is if we have a set of (models of) partial programs $Prog_i$, $i \in I$ such that (models of) valid programs can be obtained as

$$Prog_J = \bigcup_{j \in J} Prog_j$$

for certain subsets $J \subseteq I$. If we can identify the situations in which the program P_J generated by a subset J is applicable, the adaptation process roughly corresponds to context-oriented programming [47,48].

It is currently not customary to rely on explicit models of program and environment to perform the described adaptations; rather the situations in which each program or partial program is applicable are identified as the program is developed, and the reconfiguration is then performed by hard-coded program logic without resorting to a model, either by switching the strategy, or by enabling or disabling contexts.

The previous discussion suggests several interesting topics for future research: we have stated that adaptation happens when the environment changes from *Env* to *Env*′. In general it is not easy to determine *that* the environment has changed, or, if a change has occurred, what the new environment looks like. Research in this area will be able to profit from results in model-based problem solving and automated diagnosis [29]. Once a change in the environment has been identified, another challenge is to determine whether adaptation is necessary, i.e., whether $Prog, Env', Link \models Res$ still holds. For efficiency reasons it will normally not be possible to prove this formula using automated theorem proving, and more efficient ways to test the validity of this formula should be investigated for various modeling and programming paradigms.

The formalism mentioned above says nothing about useful strategies to create candidate programs for *Prog*′. One possibility is to have a fixed number of possible adaptations, or a set of building blocks for programs, as described in the examples above. Another possibility is to have certain "adaptation operators" that can modify a class of models. These operators might range from well-understood operations, such as linearly interpolating between the output values of two programs, to operations with unpredictable consequences, such as random permutations of model fragments, i.e., evolutionary algorithms operating on models. Various experiments in using evolutionary approaches to design circuits with interesting characteristic [49] and facilities for self-repair [50] have been reported. In cases where we have millions of parallel nodes, which are difficult to exploit for controlled design, the parallel evolution of several alternative solutions might deliver better results than normal design techniques. Adrian Thompson notes in [51] that evolutionary design is the only feasible approach in situations where neither the forward nor inverse model are tractable; a situation which might appear frequently in ensembles. The papers [52,53] by John

R. Koza present an optimistic opinion about the feasibility of this approach. In our estimation it is not clear whether the obtained results are representative and further research in this area is needed.

Reasoning about, or dynamically changing, the model, is only useful if the results are reflected in the behavior of the system, in other words, there has to be a *causal connection* between the internal model and the program. This implies that the program should have a stratified or meta-level architecture that can easily support these changes. While meta-level architectures have been thoroughly investigated [54] in languages such as Smalltalk or Common Lisp, and are widely used in dynamic languages such as Python, Ruby, or Groovy, many interesting problems remain. For example, the specification of meta-object protocols is still done in an ad-hoc manner, in this area ideas from the Component community might be fruitfully incorporated. Another interesting research question is how declarative meta-level reasoners can be grounded in the base program and how this grounding can be used to increase the performance of the reasoning process [55].

4.2 Limiting Adaptation and Emergence

An important consideration for adaptive systems, in particular when using evolutionary techniques but also in more controlled approaches, is how to verify that the resulting design still satisfies the specification? It is, after all, not difficult to design scenarios where interaction between different components lead to undesirable "weakly emergent" behavior.

When working with logical models it might, in some cases, be possible to prove that the result of an adaptation is correct, but in general this will be too resource intensive and unpredictable. Furthermore, this approach presupposes that the composition of well-defined local behaviors does not result in a system that exhibits undesirable behaviors on a global level. This is not generally true as, e.g., the game of Life [56] shows, where even very simple local interactions can lead to unpredictable global behavior [57,58].

The investigation of architectural patterns and invariants might prove fruitful, e.g., by combining redundancy of the evolved components with a (provably correct) controller that disables the outputs of components not fulfilling their specification. This raises a number of interesting questions regarding the adaptivity of the controller, e.g., how does the controller determine whether its specification still fulfills the requirements in the current environment?

5 Distributed Development

Ensembles are distributed systems with large-scale parallelism. As such their designers need to address many difficulties that arise from parallelism and non-determinism, such as partial failure, network latencies, problems of data replication and synchronization, and many others. For lack of space we cannot address these topics, but we are convinced that the research directions proposed

in this paper are also highly relevant for managing distribution. The web site of the FET Proactive Initiative on massive ICT systems [59] contains further information on future research directions in this area.

The distributed nature of ensembles influences not only their run time, it also plays an important role during design time. Many ensembles will be developed by independent, distributed teams, where no team has a complete view of the system, where geographically dispersed members work together on individual components and where new components can rarely be developed without taking the existing parts of the ensemble into account.

To facilitate concurrent development by independent teams, systems are usually divided into subsystems with fixed interfaces during an early stage of the system engineering process. While sub-system designers are free to develop the internals of the sub-system, the interfaces between different parts of the system are generally frozen. The expectation is that the interfaces between components change infrequently, while the division into subsystems remains fixed throughout the project.

The designers of a (sub-)system usually build several models for horizontal or vertical slices of the system; these models are commonly called *viewpoints* [60,61]. The different viewpoints of a system are not completely independent. In many cases they have certain intersections, e.g., several viewpoints may contribute functionality to the same method, or they may describe the same component in different formalisms.

The program that implements the models normally discards the information provided by the separation into different viewpoints: all viewpoints of the model are combined into a monolithic program, contributions from different viewpoints may even be intermixed in a single method-body, and in many cases there is no explicit mapping between program and model elements.

Some modern languages, such as C#, Smalltalk or Ruby allow a limited amount of separation between different viewpoints, e.g., by providing partial classes. Aspects, e.g., in AspectJ [62] or HyperJ [63] address this problem in more depth by adding mechanisms for separation of concerns or subject-oriented programming to programming languages. Our own approach [64] provides a construct called *protocols* that can express the integration of several viewpoints with partially shared functionality on the programming-language level. Aspect orientation is an active area of research; many results can be found in the proceedings of the Aspect-Oriented Software Development conferences [65].

The reasons for introducing aspect-oriented techniques remain equally valid on the system level and for modeling languages instead of programming languages [66,67,68]. In particular, often there is no single "best" decomposition of a system into subsystems but many aspects cut across subsystems. Fixing the system components and interfaces during the early design stages therefore often leads to inflexible designs that are difficult to adapt to changing customer requirements during the system life cycle [69]. Further research is needed to develop approaches that add flexibility to the structure of systems and allow

designers to postpone more decisions to later stages in the development process, when more information about the real needs of the system's users is available.

6 Improved Tools and Languages

Tools are important to suppress or simplify many of the accidental difficulties in the software development process. For large projects they are also instrumental in navigating models and source code. It is likely that the development of improved tools for programming languages such as Java has played an important role in their industry-wide adoption.

To be useful, improvements in languages have to be accompanied by support for the new features in development environments. For example, we argue that individual viewpoints that can be combined into a program can reduce the effort to develop programs. However, just defining the necessary language constructs is, in itself, not helpful. It is also necessary to provide corresponding tools that can display individual viewpoints, as well as the result of combining several or all viewpoints of a system, etc., and to integrate the new language constructs into the navigation mechanisms of the development environment, the debugger, etc.

One example where research on tools is needed is support for code and model transformations and, in particular, refactorings. Currently most refactoring tools work on the level of an abstract syntax tree; this leads to many "edge cases" where refactorings are not semantics-preserving [70,71]. Furthermore, refactorings are currently language-specific, and cannot be easily applied across abstraction layers or to program parts written in different languages. In order to gain flexibility in the system development process, first-class, semantics-based support for model and program transformation should be investigated.

This research also has connections to development tools traditionally not directly concerned with programming or modeling languages. For example, version control systems work on a purely syntactic basis; if a refactoring is transmitted between developers via the version control system, the information that a refactoring was performed is at best contained in the commit logs and tool-specific metadata, but it is not available to the repository itself. The need to understand the semantics of the program therefore extends to tools such as code repositories. This topic is closely connected to approaches for modeling change as a first-class entity, see the article [72] in this volume for a comprehensive overview. Semantics-based approaches promise particularly large benefits in distributed settings, which are becoming increasingly common because of the availability of distributed version control systems, such as [73,74,75,76].

One other area where tool support is important is in understanding and modifying systems after they have adapted. If the system model changes because the system adapted autonomously to different environmental conditions or requirements, the developer has to be able to understand the reasons for the adaptation and the consequences of the change. Furthermore, if several instances of a system are deployed, they will, in general, adapt in different ways. When updates

to these systems are deployed, they should respect these adaptations. How this can be achieved is an interesting topic for future research.

7 Domain-Specific Development

The domain of a software system plays a prominent role in almost all development approaches, e.g., in the form of requirements analysis and system models [45]. Some recent development approaches place particular emphasis on the importance of a detailed understanding of the domain [77] or problem contexts [78], but this is mostly done in an informal context.

Several development approaches try to use domain knowledge in a more formal manner. For example, software product lines [79], software factories [80], and the step-wise refinement approach of AHEAD [82,83] are approaches based on generative programming [81] which propose to develop families of related programs. This is achieved by modeling the domain and possible features of programs and by building generators that can create programs with certain combinations of features and non-functional properties from configuration specifications. While this approach has applications in certain domains, it is not yet clear whether it can be successful as a general method for developing software.

Similarly, domain-specific languages (DSLs) are sometimes claimed to facilitate the development of certain software systems by providing a simple language for specifying the problem. Here we can distinguish between *internal* DSLs where the DSL is embedded in a general-purpose programming language and *external* DSLs where the DSL is a stand-alone implementation [84]. Both variants have a long tradition in certain communities; external DSLs are common in the Unix operating system and internal DSLs have extensively been used systems built in Lisp [85]. Experience shows that while DSLs often have significant advantages there are also some problems: the behavior of DSLs is often specified by reference to the behavior of an implementation, and in particular external DSLs lead to a proliferation of languages that a developer has to understand. Techniques for precisely specifying DSLs and easily deriving an implementation from the specification remain relevant and challenging research topics.

Returning to Section 3.4, an interesting research topic is the development of theories for various application domains that can be used as deep models, or as background knowledge for deep models. These theories could be equipped with (interfaces to) reasoning components for particular aspects of the domain. For example, a theory for university management might define notions such as “university,” “student,” “lecture,” and their relationships. Associated tools might include a constraint solver for creating timetables and room assignments, or a planner for proposing courses that a student should attend. Here other research, for example in the area of the Semantic Web, might offer opportunities for collaboration.

8 Conclusions

We have presented some challenges that ensembles present to the software engineer and highlighted research directions that, in our opinion, offer opportunities for improving the state of the art in software engineering in the next 10–15 years. These research directions are often continuations of research that is already in progress. That is not a surprising situation, since currently deployed large, complicated systems already exhibit most of the characteristics of ensembles.

However, this does not mean that the problems of software engineering for ensembles have already been solved and that today's development practices are already sufficient: development projects for large systems regularly transgress the allocated budget while delivering only fractions of the planned functionality; the resulting systems invariably suffer from security holes and unforeseen failures. For example, September 16, 2008, the Munich suburban train system had to operate on an emergency program for several hours, and the main track was completely shut down for 45 minutes, because the control center was malfunctioning and had to be rebooted. In March 2008, a nuclear power plant in the USA executed an emergency shutdown after a software update was installed on a computer operating on the plant's business network [86]. In August 2008 a computer-virus was found on a laptop on board the International Space Station [87].

These examples illustrate that even the most essential, dangerous, and, hopefully, well-maintained systems can be brittle, unreliable, and insecure; software often plays an important role in their failures. It is, of course, not possible to eliminate all errors. But we should be able to expect software that is at least as robust, reliable and trustworthy as other engineered devices. We think that the research directions presented in this paper promise to take us a step closer towards this goal.

References

1. Feigenbaum, E.: Interview with Donald Knuth: A life's work interrupted. Commun. ACM 51(8), 31–35 (2008)
2. Frederick, P., Brooks, J.: No Silver Bullet Essence and Accidents of Software Engineering. Computer 20(4), 10–19 (1987)
3. Frederick, P., Brooks, J.: The Mythical Man-Month: After 20 Years. IEEE Softw. 12(5), 57–60 (1995)
4. Kranakis, E.: Primality and Cryptography. Wiley-Teubner Series in Computer Science. B. G. Teubner, Stuttgart (1987)
5. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide, 2nd edn. Addison-Wesley, Reading (2005)
6. Amelunxen, C., Schürr, A.: Vervollständigung des Constraint-basierten Assoziationskonzeptes von UML 2.0. In: Mayr, H., Breu, R. (eds.) Proc. of Modellierung 2006, Bonn, Gesellschaft für Informatik, Gesellschaft für Informatik. Lecture Notes in Informatics, vol. P-82, pp. 163–172 (2006)
7. Cook, S., Kent, S.: The unified modelling language. In: [80], pp. 611–627

8. Misra, A., Karsai, G., Sztipanovits, J.: Model-integrated development of complex applications. In: SAST 1997: Proceedings of the 5th International Symposium on Assessment of Software Tools (SAST 1997), Washington, DC, USA, p. 14. IEEE Computer Society Press, Los Alamitos (1997)
9. Karsai, G., Ledeczi, A., Neema, S., Sztipanovits, J.: The model-integrated computing toolsuite: Metaprogrammable tools for embedded control system design. In: Proc. of the IEEE Joint Conference CCA, ISIC and CACSD, Munich, Germany, pp. 50–55 (October 2006)
10. Mellor, S.J., Kendall, S., Uhl, A., Weise, D.: MDA Distilled. Addison Wesley Longman Publishing Co., Inc., Redwood City (2004)
11. Object Management Group: Model Driven Architecture (OMG) (Last visited: June 2008), http://www.omg.org/mda/
12. Schumann, J.M.: Automated Theorem Proving in Software Engineering. Springer, Heidelberg (2001)
13. CADE: Conference on Automated Deduction, Conference Web Site (last accessed 2008-09-16), http://www.cadeconference.org/
14. IJCAR: International Joint Conference on Automated Reasoning, Conference Web Site (last accessed 2008-09-16), http://www.ijcar.org/
15. Rushby, J.: Harnessing disruptive innovation in formal verification. In: Hung, D.V., Pandya, P. (eds.) Fourth International Conference on Software Engineering and Formal Methods (SEFM), Pune, India, September 2006, pp. 21–28. IEEE Computer Society, Los Alamitos (2006)
16. Gotel, O., Finkelstein, A.: An analysis of the requirements traceability problem. In: Arnold, R., Bohner, S. (eds.) Software Change Impact Analysis. IEEE Computer Society Press, Los Alamitos (1996)
17. Tekinerdogan, B., Hofmann, C., Aksit, M.: Modeling traceability of concerns in architectural views. In: AOM 2007: Proceedings of the 10th international workshop on Aspect-oriented modeling, pp. 49–56. ACM, New York (2007)
18. Kaufmann, M., Panagiotis, M., Strother, M.J.: Computer-Aided Reasoning—An Approach. Advances in Formal Methods. Kluwer Academic Publishers, Dordrecht (2000)
19. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS (LNAI), vol. 607, pp. 748–752. Springer, Heidelberg (1992)
20. Owre, S., Rushby, J., Shankar, N., Stringer-Calvert, D.: PVS: an experience report. In: Hutter, D., Stephan, W., Traverso, P., Ullman, M. (eds.) FM-Trends 1998. LNCS, vol. 1641, pp. 338–345. Springer, Heidelberg (1999)
21. McDonald, J., Anton, J.: SPECWARE—Producing Software Correct by Construction. Technical Report KES.U.01.3, Kestrel Institute (March 2001)
22. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications, 2nd edn. Cambridge University Press, Cambridge (2007)
23. Gnesi, S., Mazzanti, F.: A model checking verification environment for UML statecharts. In: Proceedings XLIII AICA Annual Conference, University of Udine - AICA 2005, October 2-5 (2005)
24. Blackburn, P., van Benthem, J.F.A.K., Wolter, F.: Handbook of Modal Logic. Studies in Logic and Practical Reasoning, vol. 3. Elsevier Science Inc., New York (2006)
25. van Harmelen, F., Lifschitz, V., Porter, B. (eds.): Handbook of Knowledge Representation. Foundations of Artificial Intelligence. Elsevier, Amsterdam (2007)

26. Brachman, R., Levesque, H.: Knowledge Representation and Reasoning. Morgan Kaufmann Publishers Inc., San Francisco (2004)
27. Sowa, J.F.: Knowledge Representation: Logical, Philosophical and Computational Foundations. Thomson Learning (1999)
28. Giarratano, J.C., Riley, G.: Expert Systems, Principles and Programming. PWS Publishing Company (2005)
29. Struss, P.: Model-Based Problem Solving. In: [25], pp. 395–465
30. Stickel, M.E.: Automated deduction by theory resolution. Journal of Automated Reasoning 1, 333–355 (1985)
31. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. 1(2), 245–257 (1979)
32. Copeland, J.B.: Cyc: A case study in ontological engineering. Electronic Journal of Analytic Philosophy 5 (1997)
33. Ramach, D., Reagan, R.P., Goolsbey, K.: First-orderized researchcyc: Expressivity and efficiency in a common-sense ontology. In: Papers from the AAAI Workshop on Contexts and Ontologies: Theory, Practice and Applications (2005)
34. Hart, P.E.: Directions for ai in the eighties. SIGART Bull. 79, 11–16 (1982)
35. Davis, R.: Expert systems: Where are we? and where do we go from here? AI Magazine 3(2), 3–22 (1982)
36. Chandrasekaran, B., Mittal, S.: Deep versus compiled knowledge approaches to diagnostic problem-solving. In: AAAI, pp. 349–354 (1982)
37. OpenCyc (last accessed 2008-09-15), http://www.opencyc.org/
38. Dutoit, A.H., McCall, R., Mistrik, I., Paech, B.: Rationale Management in Software Engineering. Springer, New York (2006)
39. Pearl, J.: Causality: Models, Reasoning and Inference. Cambridge University Press, Cambridge (2000)
40. Avin, C., Shpitser, I., Pearl, J.: Identifiability of path-specific effects. In: Kaelbling, L.P., Saffiotti, A. (eds.) IJCAI, pp. 357–363. Professional Book Center (2005)
41. McCarthy, J., Hayes, P.J.: Some philosophical problems from the standpoint of artificial intelligence. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence 4, pp. 463–502. Edinburgh University Press (1969) (reprinted in McC90)
42. Reiter, R.: Knowledge in Action. MIT Press, Cambridge (2001)
43. Mueller, E.T.: Commonsense Reasoning. Morgan Kaufmann, San Francisco (2006)
44. Hopkins, M., Pearl, J.: Causality and counterfactuals in the situation calculus. J. Log. Comput. 17(5), 939–953 (2007)
45. Sommerville, I.: Software Engineering, 8th edn. Addison-Wesley, Reading (2007)
46. Bedau, M.A.: Weak emergence. In: Tomberlin, J. (ed.) Philosophical Perspectives: Mind, Causation, and World, vol. 11, pp. 375–399. Blackwell Publishers, Malden (1997)
47. Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming: an overview of contextl. In: DLS 2005: Proceedings of the 2005 symposium on Dynamic languages, pp. 1–10. ACM, New York (2005)
48. Costanza, P., Hirschfeld, R.: Reflective layer activation in contextl. In: SAC 2007, pp. 1280–1285. ACM, New York (2007)
49. Thompson, A., Layzell, P., Zebulum, R.S.: Explorations in design space: Unconventional electronics design through artificial evolution. IEEE Trans. Evol. Comp. 3(3), 167–196 (1999)
50. Garvie, M., Thompson, A.: Scrubbing away transients and jiggling around the permanent: Long survival of FPGA systems through evolutionary self-repair. In: Metra, C., Leveugle, R., Nicolaidis, M., Teixeira, J. (eds.) ICES 2003. LNCS, vol. 2606, pp. 155–160. IEEE Computer Society, Los Alamitos (2004)

51. Thompson, A.: Notes on design through artificial evolution: Opportunities and algorithms. In: Parmee, I.C. (ed.) Adaptive computing in design and manufacture V, pp. 17–26. Springer, Heidelberg (2002)
52. Koza, J.R.: Survey of genetic algorithms and genetic programming. In: Proceedings of the Wescon 1995 - Conference Record: Microelectronics, Communications Technology, Producing Quality Products, Mobile and Portable Power, Emerging Technologies, pp. 589–594. IEEE Press, Los Alamitos (1995)
53. Koza, J.R., Keane, M.A., Yu, J., Forrest, H., Bennett, I., Mydlowec, W.: Automatic creation of human-competitive programs and controllers by means of genetic programming. Genetic Programming and Evolvable Machines 1(1-2), 121–164 (2000)
54. Kiczales, G., des Rivières, J., Bobrow, D.G.: The Art of the Metaobject Protocol. MIT Press, Cambridge (1991)
55. Hölzl, M.: A model-based architecture for entertainment applications. In: White, J. (ed.) Proceedings of the 25th International Lisp Conference, Association of Lisp Users (2005)
56. Wainwright, R.T.: Life is universal! In: WSC 1974: Proceedings of the 7th conference on Winter simulation, pp. 449–459. ACM, New York (1974)
57. Beer, R.D.: Autopoiesis and cognition in the game of life. Artif. Life 10(3), 309–326 (2004)
58. Gotts, N.M., Callahan, P.B.: Emergent structures in sparse fields of conway's game of life. In: ALIFE: Proceedings of the sixth international conference on Artificial life, pp. 104–113. MIT Press, Cambridge (1998)
59. European Commission: FET Proactive Initiative: Massive ICT Systems (last accessed 2008-09-18), http://cordis.europa.eu/fp7/ict/fet-proactive/massict_en.html
60. Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., Goedicke, M.: Viewpoints: a framework for integrating multiple perspectives in system development. International Journal of Software Engineering and Knowledge Engineering 2(1), 31–57 (1992)
61. Finkelstein, A.: Relating viewpoints. In: ACM SIGSOFT 1996 Workshop - Viewpoints 1996, p. 157. ACM Press, New York (1996)
62. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
63. Ossher, H., Tarr, P.: Using multidimensional separation of concerns to (re)shape evolving software. Commun. ACM 44(10), 43–50 (2001)
64. Hölzl, M.: Combining language extensions. In: de Lacaze, R. (ed.) Proceedings of the International Lisp Conference 2003, Association of Lisp Users (2003)
65. AOSD: Aspect-oriented software development conference web site (last accessed 2008-09-15), http://aosd.net/
66. Lahire, P., Morin, B., Vanwormhoudt, G., Gaignard, A., Barais, O., Jézéquel, J.M.: Introducing variability into aspect-oriented modeling approaches. In: [88], pp. 498–513
67. Whittle, J., Moreira, A., Araújo, J., Jayaraman, P.K., Elkhodary, A.M., Rabbi, R.: An expressive aspect composition language for uml state diagrams. In: [88], pp. 514–528
68. Zhang, G., Hölzl, M.M., Knapp, A.: Enhancing uml state machines with aspects. In: [88], pp. 529–543
69. Poppendieck, M., Poppendieck, T.: Lean Software Development: An Agile Toolkit. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)

70. Kegel, H., Steimann, F.: Systematically refactoring inheritance to delegation in java. In: Robby (ed.) ICSE, pp. 431–440. ACM, New York (2008)
71. Steimann, F.: Personal communication
72. Nierstrasz, O., Denker, M., Gîrba, T., Lienhard, A., Röthlisberger, D.: Change-enabled software systems. In: Wirsing, M., et al. (eds.) Software-Intensive Systems. LNCS, vol. 5380. Springer, Heidelberg (2008)
73. Git: Web site for the git distributed version control system (last accessed 2008-09-09), http://git.or.cz/
74. Mercurial: Web site for the mercurial distributed version control system (last accessed 2008-09-09), http://www.selenic.com/mercurial/wiki/
75. Monotone: Web site for the monotone distributed version control system (last accessed 2008-09-09), http://monotone.ca/
76. Darcs: Web site for the darcs distributed version control system (last accessed 2008-09-09), http://darcs.net/
77. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, Reading (2004)
78. Jackson, M.: Problem Frames: Analysing and Structuring Software Development Problems. Pearson Education, London (2001)
79. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Heidelberg (2005)
80. Greenfield, J., Short, K.: Software Factories: Assembling Applications with Patterns, Frameworks, Models and Tools. Wiley, Chichester (2004)
81. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools and Applications. Addison-Wesley Professional, Reading (2000)
82. Rauschmayer, A., Knapp, A., Wirsing, M.: Consistency checking in an infrastructure for large-scale generative programming. In: Proc. 19th IEEE Int. Conf. Automated Software Engineering (ASE), September 2004, pp. 238–247. IEEE, Los Alamitos (2004)
83. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling step-wise refinement. ACM Transactions on Software Engineering (TSE) 30(6), 355–371 (2004)
84. Fowler, M.: Domain specific languages (last accessed 2008-09-16), http://martinfowler.com/dslwip/
85. Graham, P.: On Lisp: advanced techniques for Common Lisp. Prentice-Hall, Englewood Cliffs (1994)
86. Krebs, B.: Cyber incident blamed for nuclear power plant shutdown. The Washington Post (June 5, 2008) (last accessed 2008-09-18), http://www.washingtonpost.com/wp-dyn/content/article/2008/06/05/AR2008060501958.html
87. Powell, D.: Space station computer virus raises security concerns. New Scientist Space (August 29, 2008) (last accessed 2008-09-18), http://space.newscientist.com/article/dn14628-space-station-computer-virus-raises-security-concerns.html
88. Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.): MODELS 2007. LNCS, vol. 4735. Springer, Heidelberg (2007)

Change-Enabled Software Systems

Oscar Nierstrasz, Marcus Denker, Tudor Gîrba,
Adrian Lienhard, and David Röthlisberger

Software Composition Group, University of Bern
http://scg.unibe.ch/

Abstract. Few real software systems are built completely from scratch nowadays. Instead, systems are built iteratively and incrementally, while integrating and interacting with components from many other systems. Adaptation, reconfiguration and evolution are normal, ongoing processes throughout the lifecycle of a software system. Nevertheless the platforms, tools and environments we use to develop software are still largely based on an outmoded model that presupposes that software systems are closed and will not significantly evolve after deployment. We claim that in order to enable effective and graceful evolution of modern software systems, we must make these systems more amenable to change by (i) providing explicit, first-class models of software artifacts, change, and history at the level of the platform, (ii) continuously analysing static and dynamic evolution to track emergent properties, and (iii) closing the gap between the domain model and the developers' view of the evolving system. We outline our vision of dynamic, evolving software systems and identify the research challenges to realizing this vision.

1 Introduction

Software inevitably changes, but our development methods, programming languages, development environments and run-time systems generally assume that one is building a closed, internally consistent application, which will not significantly change after deployment. Anticipated evolution can be built in to some extent, for example by applying well-known design patterns, but unanticipated changes in requirements are hard to accommodate without reengineering the system, redeploying it, and possible migrating persistent data.

The vision of an *eternal software-intensive system* is that of a system that can survive such unanticipated changes with little or no human intervention at the lowest level [59]. We claim that this vision can only be realized if *software change is enabled* in a fundamental way in our platforms, run-time environments and development environments [42]. In particular, not only software systems themselves, but their development and support environments need to be far more *dynamic* than they are today. Specifically, what does this entail?

– First of all, we need to provide *platforms* in terms of programming languages and run-time environments that make it possible to *manipulate and operate*

M. Wirsing et al. (Eds.): Software-Intensive Systems, LNCS 5380, pp. 64–79, 2008.

on change as a first-class entity. This in turn implies that an evolving software system is not only model-driven, but actually "self-aware" — it must have a first-class representation of itself available to enable change. To control the scope of change, change itself should be represented as a first-class, high-level entity. To manage change over time, the history of the system must also be accessible and first-class (see Section 2).
- Second, an evolving software system must be *capable of analyzing itself*, and in particular of recognizing *emergent properties*. This means that the evolution of the static and dynamic models must be monitored, and the resulting data be analyzed as the system is running (see Section 3).
- Third, to enable continuous evolution, a software system must *close the gap between the development and deployment views* of itself. Domain models, usage models, and features, for example, must be made explicit in the system to facilitate change (see Section 4).

We will explore these themes in some detail, in each case summarizing previous work, and establishing a research agenda for further work. We conclude with summary remarks about next steps.

2 Self-aware Platforms to Support Change

Traditionally, the development and deployment of software are viewed as being separate in time and space: first a system is developed, then it is deployed. Indeed, in the classical view, we deal with two completely different artifacts: the source code that can be developed, debugged and understood on the one hand, and on the other hand a generated, closed, non-understandable binary program that just can be run.

Classical development plays out like a finite game with fixed rules and boundaries. Evolving software systems, on the other hand, are better thought of as *infinite games* without fixed rules or boundaries [5]. Evolving systems will not have a clear separation of development and deployment. The system will continue to evolve when it is already deployed. The systems of the future will not be developed from the outside as a finite game. Development itself will be part of the infinite game of the system. Evolution needs to happen in parts of the system, while it is running.

We cannot afford to stop and restart a continuously evolving software system, just as we cannot stop and restart the Internet. The Internet has been up and running since 1969, although many of its atoms have been changed many times since then. The software intensive systems of the future will need to learn from these loosely-coupled, long-lived systems. To support this view, we need appropriate core technologies in terms of programming languages and run-time systems that can serve as a platform for developing evolving software systems.

2.1 Previous Work

In order to enable change at run-time, an evolving system must be able to fully reflect on itself, that is, it must be *self-aware*. It is not enough to be

model-driven. The models must be explicit and accessible to the run-time system. A *reflective system* provides a description of itself available from within. This description can be queried (*introspection*) as well as changed (*intercession*). In the past, computational reflection has been an active area of research [13,15,41,55]. Nevertheless, languages used in industry today do not provide full reflection, and many mainstream languages have no self representation at all (for example C and C++). More recently created languages like Java and C# support limited introspection, but no intercession.

In recent years, dynamic languages have attracted much attention [43] and are increasingly being used in industry (*e.g.*, Python, Ruby, Smalltalk). Dynamic languages provide a representation that can be queried and changed at run-time and thus are reflective. But even in these dynamic languages, the support for reflection is limited. The *structural* representation of the program stops at the granularity of the method. Classes and methods are represented as objects and available to be queried and changed, but the structure of the methods themselves is not represented. *Behavioral* reflection is limited as we cannot change behavior on a fine-grained level. In addition, when applying behavioral reflection to the system (as opposed to an application), the programmer will soon run into the problem of meta-object call recursion [11].

We have extended structural reflection to model the structure of methods: sub-method reflection [8] represents the complete structure, down to the code itself. The representation can be annotated and thus can be used for integrating tools. One example is feature annotation [10]. Instead of recording full traces to analyze features, we can simply annotate the static structure of the system with feature information. Partial behavioral reflection [51,58] provides means to select where and when a meta-object is activated and allows us to define which information is passed to the meta-object. We can introduce behavioral changes at run-time which provides the basis to supporting unanticipated change to the systems. Examples range from tools like tracers or profilers [51] to changes of the language semantics, for example transactional memory [47]. The problem of meta-object call recursion is solved by representing meta-level execution as a context and by making meta-object activation context-aware [11].

It is well-established that suitable abstractions are needed to enable programming in the large [45]. But in the case of scale, we need to think again: are existing abstractions good enough for *very large* software systems? For example, as software gets larger the assumption that every part of the system must stay in sync with every other part is not very convincing because the systems of the future will be so large that we will never be able to evolve them in a single, synchronous step. As a consequence, an evolving system must be able to cope with multiple, inconsistent views of itself.

Inconsistency is only tolerable if specific and individual views appear to be locally consistent. Instead of allowing all changes to be globally visible, we need a means to control the scope of changes. That is, evolving systems must support a notion of *context* and the run-time infrastructure must be *context-aware*. Being able to dispatch on context means that we need to support a form of

context-oriented programming [7,27]. Visibility of changes can then be restricted to the context in which these changes are guaranteed to be valid.

Changeboxes [9] provide a mechanism for encapsulating change as a first-class entity in a running software system. Changeboxes support multiple, concurrent and possibly inconsistent views of software artifacts within the same running system. Since changeboxes are first-class, they can be manipulated to control the scope of change in a running system. Furthermore, changeboxes capture the semantics of change. Changeboxes can be used, for example, to encapsulate refactorings, or to replay or analyze the history of changes.

2.2 Research Agenda

We maintain that both *reflection* and *context* are crucial to support change and evolution. There has been much recent progress, but more research is needed. In particular, the key ideas emerging from previous research need to be consolidated and integrated into a comprehensive model.

Efficient and Practical Reflective Languages. Sub-method structural reflection and partial behavioral reflection are improvements over conventional reflective systems. One problem with reflection, even with efficient partial reflection [51,58] is performance. With behavioral reflection, we introduce new behavior that replaces the default behavior. One example is method lookup. The default lookup is extremely optimized and realized in the virtual machine, so any reflective redefinition is often slower by an order of magnitude. This difference in performance can render a system unusable in practice. We need to integrate reflection better into the virtual machine, leveraging the dynamic code generator of modern just-in-time compilers. Another interesting question is how to resolve static typing with reflection. Type-systems reason about properties that can be guaranteed in the future, whereas reflection is about changing the future. We need a way to check reflective change before it is applied to the system.

Backward Compatibility. Backward compatibility is the enemy of forward evolvability. Nevertheless, we cannot live in a world where the old is ignored. An often overlooked property of software is that new systems can simulate the old, and the recent trends in hardware virtualization have shown that simulation of the old is far easier than for the new to stay compatible. A snapshot of an old Windows machine can run on a virtual machine forever, whereas keeping an operating system compatible forever is bound to fail. Programming languages for evolving systems should provide backwards compatibility in the same way: we need a first class description of the history of all code of the system, freeing the present from being compatible with the past while at the same time providing the possibility to go back in time easily. The system should provide complete, runnable snapshots of itself at any point in the past. Our work on changeboxes forms one first step towards this goal. However, changeboxes are only concerned with code, thus an important aspect of future work is the problem of migrating data between versions.

Contextual Reflection. Context appears to be deeply related to reflection. In the case of changeboxes, the currently active changebox provides a context for execution. For partial behavioral reflection, we solve the problem of meta-object call recursion by representing the execution of the meta-level as a context [11]. We think that the next step is to revisit these cases and integrate the notion of context into the reflective model of the language and the virtual machine.

Languages Supporting Multiple Views. In general, we need to explore multi-dimensional object systems. After achieving a reflective model that is aware of context, the next step is to build language support that makes this concept available to the programmer. Some very relevant work has been done in the past, for example the work on PIE [4,22,23,24], Us [56] and more recently ContextL and ContextS [7,27]. More research is needed to explore how to combine these ideas with contextual reflection and changeboxes.

Evolving Languages. Evolving systems need languages that support continuous development and evolution. But there is another aspect when considering the language itself: to think that we can envision the perfect language to realize all future systems is to treat language design like a finite game. Thus a language suited for implementing ever-evolving software systems needs to be *itself* an evolving system. An evolving language must evolve to incorporate new ideas and practices while it is used. It needs to be extensible and growable from within [57].

3 Monitoring and Analyzing Change

To change a system we must first understand the system and the consequences of change. Since change inevitably causes the system to drift from its initial documentation, the most reliable source of information is the system itself. A self-aware system can reflect on its own specification, which is an aid to static analysis. But emergent properties as well as program failures can only be monitored with the help of dynamic analysis [26].

Ideally, a productive system should constantly monitor and analyze itself. This would allow us to discover properties that are only visible over a longer period of time, such as performance degradation, memory leaks, shifts in how the system is used, effects of structural changes etc. Furthermore, collecting detailed data about the program execution can provide crucial information to uncover the cause of program failures.

3.1 Previous Work

Program failures for large, long-lived software systems can be hard to reproduce, and hard to simulate in a test environment. As a consequence, systems need to dynamically analyze their behavior to gather execution history while they are running and allow for remote and safe debugging inside the live system. To be able to reason about the run-time behavior of a system, dynamic information has to be

linked with the model of the software structure. A platform that provides an integrated, high-level model of its own structure, design, and behavior (see Section 2) would offer an appropriate foundation for building self-analyzing systems.

The challenge we face with such an architecture is that the current state of the art in dynamic analysis does not permit run-time information to be gathered below the method level in live systems due to performance reasons. The main obstacles are (i) code instrumentation requires a system to be restarted, (ii) run-time overhead can be huge (up to a factor of 100 or more for non-trivial programs [34]), and (iii) available memory to store the gathered data limits the analysis to only few and short user sessions.

Most existing dynamic analysis approaches are detached from the run-time environment, *i.e.*, virtual machine, and hence have only limited means to adapt and reconfigure the system at runtime without restarting it. Instrumentation code is weaved into the application code at compile time, for instance through bytecode manipulation. This additional code then generates data during execution, which is either stored as a trace of events in memory or in a database where it is processed *post mortem*. An evolving system that needs to be running all the time cannot be restarted to reconfigure the analyzer. Therefore, the run-time analysis of a self-aware system needs to be an integral part of its run-time environment. Like this, no hard-wired instrumentation code is required, but rather the analyzer is a self-aware component of the virtual machine. The analyzer needs to be always running and capable of adjusting its own behavior, much like garbage collectors are always active in modern virtual machines.

We have addressed some of these problems as follows [37,39,38]. We have extended the object memory model of conventional object-oriented virtual machines by representing object references as real objects on the heap. In this way we seamlessly integrate historical execution data into the object model of the virtual machine. Our approach discards unneeded historical execution data by employing the standard garbage collector of the VM. We showed that this approach can dramatically improve the data explosion problem and has much lower execution overheads compared to other back-in-time debuggers [34,46] that are not implemented at the virtual machine level.

We have extended the common dynamic analysis model with the notion of *object aliases*. That is, object references are explicitly represented in our model, which allows us to track the flow of objects in the system or to analyze how side effects are produced [40,38].

By enabling dynamic analysis on live systems, innovative debugging and analysis techniques come within reach. In a recent study, Liblit *et al.* examined bug symptoms for various programs and found that in 50% of the cases the execution stack contains essentially no information about the bug's cause [35]. Back-in-time debuggers [28,34,46] allow the developer to explore the program state at those points in time that are no longer represented on the run-time stack. We are currently exploring the development of a highly performant back-in-time debugger on top of our history-aware VM.

In evolving software systems, the changes to the static parts are directly accessible as first class entities. As such, in evolving software systems, not only the run-time is dynamic, but also the static part is dynamic when seen from an historical perspective. Treating history as a first-class entity enables analyses of the evolution of software artifacts [17]. For example, we can predict where changes are likely to occur [19], we can detect classes that are changed frequently [21], or we can identify crosscutting concerns by detecting which parts change at the same time and in the same way [18].

Given the size of evolving systems, they will not be developed by an isolated team, but rather by several teams that are physically distributed. In this context, the social aspect of the development will become increasingly important [6]. Thus, analysis should also include reasoning about how developers collaborate [2,20,25].

3.2 Research Agenda

In the long-term, we expect that run-time monitoring of program execution and evolution will become not only practical but essential to the survival of long-lived software systems. To make this a reality however, further research will be needed in the following areas.

Efficient Run-time Analysis. Dynamic analysis will only be widely adopted if it is cheap in time and space. The emergence of multi-core architectures for off-the-shelf desktop and laptop machines suggests that parallelization should be explored as one way to reduce the execution time. Even though future hardware capacity will allow for storing more data more efficiently, more advanced models for discarding unneeded data are important, as faster running systems also produce data faster.

Detecting Emergent Properties. High-level run-time models are needed to reason about the system from within the system. In an object-oriented system, recording method execution events alone is not enough for certain analyses and for debugging. Also, a run-time model should be seamlessly connected with the static model, which captures structural program entities and their evolution. This allows for recognizing links between the behavior of a system and its evolution and can serve as a source of information for maintainers in their development environment. By correlating static and dynamic information over time, one should be able to detect emergent properties, such as maintenance hot spots, performance bottlenecks and other opportunities for refactoring and reengineering.

Automatic Model Reconstruction. Heterogeneous and legacy sources of information can be obstacles to analysis and evolution. Yet another complicating factor is the use of different programming languages and media such as mainstream compiled languages, scripting languages, domain specific languages, HTML, XML and query languages within the same system. Furthermore, some of the languages used will be either legacy languages or dialects (such as legacy dialects of C++). Post-hoc parsing of components built with these languages will be

difficult and error-prone since the original language specifications may not be available. Thus, an evolving software can be seen as a multi-dimensional space of data that needs to be continuously analyzed. Techniques will be needed to automatically reconstruct high-level models and meta-models from lower-level data, without necessarily having up-to-date access to the syntax specifications.. Possible approaches include abstraction from examples [44] and formal concept analysis [1,36], amongst many others.

4 Enabling Change for the Developer

The evolution, or rather continuous development, of evolving systems places special demands on the development environment. To some extent systems can be designed for evolution. But if we see the development of an evolving system as an infinite game, it becomes clear that one cannot anticipate all forms of evolution.

Current IDEs focus on providing the developer only with a static view of the source code without offering any information about how the code is actually executed at run-time, about why a bug occurred, or about whether there are performance issues or memory consumption problems. Furthermore, modern IDEs do not provide means to bridge the gap between the users' view and the developers' view of the system. For instance, it is hard to locate and understand a specific feature of a large system by studying its sources code alone.

We envision a development environment in which change is enabled by bridging the static and dynamic views of the system and by bringing the results of dynamic analysis to the IDE. Ultimately, the IDE should be an active player in the development process, enabling developers to interactively manipulate and extend the system at a high level of abstraction.

4.1 Previous Work

Support for refactoring, reorganizing and reengineering must be part of the evolving system. The state-of-the-art in refactoring support is still in its infancy [16]. Many modern IDEs provide some automated mechanisms to change and evolve software systems but they tend to be low level, like renaming a class or moving a method [49]. Furthermore, developers receive little guidance in identifying opportunities for refactoring [33], and the knowledge about performed refactorings is usually not kept. A promising approach is to offer a better versioning system that is able to store the high level knowledge about a change [48] and then provide this information for further analysis [12].

Empirical studies report that a developer performing maintenance tasks on a system spends at least 35% of the time in navigating source code [31]. A maintenance-oriented IDE should present the developer with a working set of source code containing all functionality for a specific maintenance task to reduce the navigational load. By monitoring the programmer's activity to get a degree-of-interest for program elements scattered across a large code base, the IDE can

reveal code elements that are likely to be important for the task at hand [30]. Other proposals [50,54] focus on emphasizing relations between source artifacts by recommending related artifacts based on the past sequence of browsing, or by characterizing the kinds of changes that have taken place during development sessions [48].

Hermion is an experimental IDE that brings run-time information to the developer to better support maintenance tasks [53]. Dynamic information gathered at run-time provides the developer with the possibility to directly navigate to actual senders and implementors of messages, or to navigate to the actual concrete types to which variables have been dynamically bound. Statistical data about the run-time call graph is also integrated into the static code views, and is dynamically updated as the code under development is further exercised. Dynamic monitoring is realized by means of mechanisms in the underlying reflectivity framework (see Section 2) which support unanticipated partial behavioural reflection [51]. This allows running code to be instrumented on-the-fly at a high level of abstraction, and without modifying the underlying source code.

Visualizations can convey complex information in a condensed and effective manner. Extensions to the VisualWorks Smalltalk IDE (*e.g.*, RBCrawler) integrate various well-known visualizations such as class blueprints [14], polymetric views [32] or system complexity views [32]. They are well integrated in the static source navigation tools of this IDE, however, they do not take dynamic information into account and are thus just a starting point for further work.

One particularly useful application of visualization is to correlate static views of software components, with the features of the running systems. Feature-driven browsing exploits run-time information gathered when exercising features and presents this information as a visual map [52]. This enables us to quickly identify the components responsible for a feature and to highlight the dependencies between different features. As this feature visualization is integrated in the IDE, the developer can also better assess the impact of a source code level change on various features of the system if there is an explicit mapping between features and source artifacts available.

4.2 Research Agenda

Today's IDEs are largely passive players in the development process, though there is a clear and gradual trend towards IDEs and tools that play a more active role. We believe that further research is needed here to make the IDE a more active player in enabling change. Some promising research tracks follow.

Automatic and Continuous Execution. Not only should the IDE provide complete information, it should also do this efficiently to avoid slowing down the development process. To achieve efficient integration of dynamic information, we envision applying the concept of continuous integration by running the system automatically and continuously in the background. This relieves the developer from the burden of having to manually trigger the execution of the system while modifying it.

Full Coverage. Since manually triggered dynamic analyses normally do not cover the entire system (*i.e.*, all system's features), the IDE should assume the responsibility to ensure full coverage and cover all code still being used by the application. The IDE should analyze all execution paths through the system by running the code currently being studied by the developer. If system monitoring and analysis (see Section 3) can be made sufficiently efficient and non-intrusive, possibly by exploiting the possibilities of parallel hardware, information gathered from the execution of the live system by real users and certainly also of recorded scripts could also be fed automatically back to the IDE.

Exploiting Fine-grained Change Histories. Software is currently developed following a checkout/change/commit life cycle. This approach hides the local events and changes from the overall development. To limit this loss of information developers are advised to commit as often as possible in the central repository. In the future we envision a central environment that is tightly integrated with the versioning system and that stores all changes performed on the system. Furthermore, instead of being snapshot-based, change logs will capture the *intent* of changes, such as common refactoring operations. This information can then be fed back into the IDE so that developers can immediately access historical information in a productive and integrated way.

Autonomous System Evolution. The IDE as an active player should be able to autonomously perform many maintenance tasks. For instance, the IDE should be able to automatically detect defects, and with the help of dynamic analysis, suggest repair strategies to the developer. If a certain feature of such an evolving system is broken, the environment should be able to locate this feature in source code and determine which classes or methods need to be corrected to successfully solve the issue. Fine-grained historical information about changes to the software supporting the defective features should also be a valuable input to automatically provide focus to the developer to correct the defect.

Model-Centric Development. Software developers today generally edit and manipulate textual representations of the systems being developed, *i.e.*, as source code. However, text is by definition static while the resulting system is dynamic and neither knows nor cares about the textual representation from which it has initially been built. We envision a system in which the developer directly manipulates high-level models that more closely represent the conceptual entities of the software application. Such an approach would be *model-centric* rather than "model-driven", since models would directly represent the running application, rather than serving to generate it. Such a system could be realized by a visual programming environment that enables developers to develop and change directly the dynamic structure, *e.g.*, by working with visual components of a system's behavior that can be dragged, dropped, parameterized, adapted, extended and reorganized, directly modifying the behavior.

5 Concluding Remarks

We have argued that traditional approaches to software development do not adequately support the inevitable change that software will undergo during its lifetime. We have summarized some current and ongoing research activities to enable change for long-lived software systems, and we have presented an agenda of critical research topics for further investigation. Briefly, we see the need for further research in the following three related areas:

- *Self-aware platforms to enable change.* In order to enable change for software applications, the underlying platform (*i.e.*, programming language and run-time environment) must support change in a deep way. Such systems must therefore be *reflective*, by means of *high-level models*. Change must be modeled as a *first-class entity*, so that changes can be manipulated and reasoned about. Finally, the platform must be *context-aware* so that the scope of change can be controlled to ensure safe evolution at run-time. We have argued that these ingredients are necessary for a future generation of self-aware, long-lived, evolving software applications. Some of these ideas have been prototyped, but much research is needed before they are ready to be adopted in mainstream platforms.
- *Monitoring and analyzing change.* A long-lived, evolving software system must provide means to monitor and analyze change, both in terms of short-term, dynamic adaptations, and long-term software evolution. Dynamic analysis today is mostly *post hoc*. For performance reasons it is generally impractical to monitor and analyze live systems. Novel techniques for monitoring, debugging and analyzing live systems are being developed, but further research is needed to make these approaches practical, for example by exploiting the possibilities of the emerging class of multi-core processors. Further research is also needed to detect emergent properties of long-lived evolving software systems, particularly in terms of correlating static and dynamic software structures. Automatic detection of maintenance hot spots, performance bottlenecks, and opportunities for refactoring are some of the areas where further research is needed.
- *Enabling change for the developer.* Modern application development environments essentially provide developers with static views of the application source code. Particularly in the case of object-oriented software development, the gap between the static and the run-time structures implies that much valuable information is simply not available to the developer. By bringing together static and dynamic views of the live system, many opportunities present themselves to improve the productivity of the developer. For instance, dynamic information integrated into the static software views provides richer navigation possibilities. Further research is needed to make the IDE itself an active player that exercises, tests and analyzes the live system as it is being developed, actively updates the static views with the results of dynamic analysis, automatically detects defects and actively suggests repair strategies, and keeps track of changes and their intent at a finer granularity than is currently supported. As a long-term goal, we envision IDEs that

are not so much focussed on editing source code, but rather support direct manipulation and transformation of *models* of the live system under development.

As a closing remark, we note that this kind of research is almost impossible to carry out effectively by isolated researchers. A certain critical mass in terms of software infrastructure is needed before research results stabilize and new ideas can be built on top of older ones. Much of current research results in prototypes that do not even begin to achieve the level of stability that is needed before other researchers can build something new on top of them.

On the one hand, we advocate that the research process will need to acknowledge and to reward the engineering effort. In the future, research and engineering must meet to face the wide space opened by evolving systems. On the other hand, just like evolving systems will not be the result of one team's work, we advocate that research will need to break the group boundary and open towards research networks [3].

Bringing together research and engineering will also bring together two worlds that are now rather separated: research and practice. Practitioners face real problems and need new ideas to solve these problems, but cannot afford the time and effort to experiment with unproven ideas. Researchers need real problems to develop new ideas, but cannot afford the effort to fully validate and mature these ideas in a practical setting. Each group is under pressure to get their products out the door with acceptable quality and minimum cost. As a consequence few new ideas get proven in practice, and real problems of practitioners tend not to propagate in the research environment. The perceived cost of collaboration is just too high.

A first step to bring these groups together and reduce the cost of collaboration is to provide an infrastructure in which new ideas can be quickly implemented, tested and adopted. The need for collaboration to build a successful infrastructure can be seen in the wide adoption of Eclipse as a platform [29]. Many teams contribute to Eclipse due to its open architecture, and many researchers are using it for implementing their vision. While Eclipse is not an academic exercise, it does facilitate software evolution research. To facilitate relevant and collaborative research into evolving software intensive systems, an analogous common infrastructure will be needed upon which both research and practice can build.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008) and that of the Hasler Foundation for the project "Enabling the evolution of J2EE applications through reverse engineering and quality assurance".

References

1. Arévalo, G., Buchli, F., Nierstrasz, O.: Detecting implicit collaboration patterns. In: Proceedings of WCRE 2004 (11th Working Conference on Reverse Engineering), pp. 122–131. IEEE Computer Society Press, Los Alamitos (2004)
2. Balint, M., Gîrba, T., Marinescu, R.: How developers copy. In: Proceedings of International Conference on Program Comprehension (ICPC 2006), pp. 56–65 (2006)
3. Bennis, W., Biederman, P.W.: Organizing Genius — The Secrets of Creative Collaboration. Perseus Books (1997)
4. Bobrow, D.G., Goldstein, I.P.: Representing design alternatives. In: Proceedings of the Conference on Artificial Intelligence and the Simulation of Behavior (July 1980)
5. Carse, J.P.: Finite and Infinite Games — A Vision of Life as Play and Possibility. Ballantine Books (1987)
6. Conway, M.E.: How do committees invent? Datamation 14(4), 28–31 (1968)
7. Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming: An overview of ContextL. In: Proceedings of the Dynamic Languages Symposium DLS 2005, co-organized with OOPSLA 2005, pp. 1–10. ACM, New York (2005)
8. Denker, M., Ducasse, S., Lienhard, A., Marschall, P.: Sub-method reflection. Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007 6(9), 231–251 (2007)
9. Denker, M., Gîrba, T., Lienhard, A., Nierstrasz, O., Renggli, L., Zumkehr, P.: Encapsulating and exploiting change with Changeboxes. In: Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007), pp. 25–49. ACM Digital Library (2007)
10. Denker, M., Greevy, O., Nierstrasz, O.: Supporting feature analysis with runtime annotations. In: Proceedings of the 3rd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2007), pp. 29–33. Technische Universiteit Delft (2007)
11. Denker, M., Suen, M., Ducasse, S.: The meta in meta-object architectures. In: Proceedings of TOOLS EUROPE 2008. LNBIP, vol. 11, pp. 218–237. Springer, Heidelberg (2008)
12. Dig, D., Manzoor, K., Johnson, R., Nguyen, T.: Refactoring-aware configuration management for object-oriented programs. In: International Conference on Software Engineering (ICSE 2007), pp. 427–436 (2007)
13. Ducasse, S.: Evaluating message passing control techniques in Smalltalk. Journal of Object-Oriented Programming (JOOP) 12(6), 39–44 (1999)
14. Ducasse, S., Lanza, M.: The class blueprint: Visually supporting the understanding of classes. Transactions on Software Engineering (TSE) 31(1), 75–90 (2005)
15. Ferber, J.: Computational reflection in class-based object-oriented languages. In: Proceedings OOPSLA 1989, ACM SIGPLAN Notices, vol. 24, pp. 317–326 (October 1989)
16. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison Wesley, Reading (1999)
17. Gîrba, T., Ducasse, S.: Modeling history to analyze software evolution. Journal of Software Maintenance: Research and Practice (JSME) 18, 207–236 (2006)
18. Gîrba, T., Ducasse, S., Kuhn, A., Marinescu, R., Raţiu, D.: Using concept analysis to detect co-change patterns. In: Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2007), pp. 83–89. ACM Press, New York (2007)

19. Gîrba, T., Ducasse, S., Lanza, M.: Yesterday's Weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In: Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM 2004), pp. 40–49. IEEE Computer Society, Los Alamitos (2004)
20. Gîrba, T., Kuhn, A., Seeberger, M., Ducasse, S.: How developers drive software evolution. In: Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2005), pp. 113–122. IEEE Computer Society Press, Los Alamitos (2005)
21. Gîrba, T., Lanza, M., Ducasse, S.: Characterizing the evolution of class hierarchies. In: Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR 2005), pp. 2–11. IEEE Computer Society, Los Alamitos (2005)
22. Goldstein, I.P., Bobrow, D.G.: Descriptions for a programming environment. In: Proceedings of the First Annual Conference of the National Association for Artificial Intelligence (August 1980)
23. Goldstein, I.P., Bobrow, D.G.: Extending object-oriented programming in Smalltalk. In: Proceedings of the Lisp Conference, pp. 75–81 (August 1980)
24. Goldstein, I.P., Bobrow, D.G.: A layered approach to software design. Technical Report CSL-80-5, Xerox PARC (December 1980)
25. Greevy, O., Gîrba, T., Ducasse, S.: How developers develop features. In: Proceedings of 11th European Conference on Software Maintenance and Reengineering (CSMR 2007), pp. 256–274. IEEE Computer Society, Los Alamitos (2007)
26. Hamou-Lhadj, A., Lethbridge, T.: A survey of trace exploration tools and techniques. In: Proceedings IBM Centers for Advanced Studies Conferences (CASON 2004), Indianapolis IN, pp. 42–55. IBM Press (2004)
27. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology 7(3) (March 2008)
28. Hofer, C., Denker, M., Ducasse, S.: Design and implementation of a backward-in-time debugger. In: Proceedings of NODE 2006. Lecture Notes in Informatics, vol. P-88, pp. 17–32. Gesellschaft für Informatik, GI (September 2006)
29. Holzner, S.: Eclipse. O'Reilly, Sebastopol (2004)
30. Kersten, M., Murphy, G.C.: Mylar: a degree-of-interest model for ides. In: AOSD 2005: Proceedings of the 4th international conference on Aspect-oriented software development, pp. 159–168. ACM Press, New York (2005)
31. Ko, A.J., Aung, H., Myers, B.A.: Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In: ICSE 2005: Proceedings of the 27th international conference on Software engineering, pp. 125–135 (2005)
32. Lanza, M., Ducasse, S.: Polymetric views—a lightweight visual approach to reverse engineering. Transactions on Software Engineering (TSE) 29(9), 782–795 (2003)
33. Lanza, M., Marinescu, R.: Object-Oriented Metrics in Practice. Springer, Heidelberg (2006)
34. Lewis, B., Ducassé, M.: Using events to debug Java programs backwards in time. In: OOPSLA Companion 2003, pp. 96–97 (2003)
35. Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I.: Scalable statistical bug isolation. In: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2005), pp. 15–26. ACM, New York (2005)
36. Lienhard, A., Ducasse, S., Arévalo, G.: Identifying traits with formal concept analysis. In: Proceedings of 20th Conference on Automated Software Engineering (ASE 2005), pp. 66–75. IEEE Computer Society, Los Alamitos (2005)

37. Lienhard, A., Ducasse, S., Gîrba, T.: Taking an object-centric view on dynamic information with object flow analysis. Journal of Computer Languages, Systems and Structures (to appear, 2008)
38. Lienhard, A., Gîrba, T., Greevy, O., Nierstrasz, O.: Test blueprints – exposing side effects in execution traces to support writing unit tests. In: 12th European Conference on Software Maintenance and Reengineering (CSMR 2008), pp. 83–92. IEEE Computer Society, Los Alamitos (2008)
39. Lienhard, A., Gîrba, T., Nierstrasz, O.: Practical object-oriented back-in-time debugging. In: 22nd European Conference on Object-Oriented Programming (ECOOP 2008). LNCS, vol. 5142, pp. 592–615. Springer, Heidelberg (2008)
40. Lienhard, A., Greevy, O., Nierstrasz, O.: Tracking objects to detect feature dependencies. In: Proceedings International Conference on Program Comprehension (ICPC 2007), Washington, DC, USA, pp. 59–68. IEEE Computer Society, Los Alamitos (2007)
41. Maes, P.: Computational Reflection. PhD thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Brussels Belgium (January 1987)
42. Nierstrasz, O.: Software evolution as the key to productivity. In: Wirsing, M., Knapp, A., Balsamo, S. (eds.) RISSEF 2002. LNCS, vol. 2941, pp. 274–282. Springer, Heidelberg (2004)
43. Nierstrasz, O., Bergel, A., Denker, M., Ducasse, S., Gaelli, M., Wuyts, R.: On the revival of dynamic languages. In: Gschwind, T., Aßmann, U. (eds.) SC 2005. LNCS, vol. 3628, pp. 1–13. Springer, Heidelberg (2005)
44. Nierstrasz, O., Kobel, M., Gîrba, T., Lanza, M., Bunke, H.: Example-driven reconstruction of software models. In: Proceedings of Conference on Software Maintenance and Reengineering (CSMR 2007), pp. 275–286. IEEE Computer Society Press, Los Alamitos (2007)
45. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. C. ACM 15(12), 1053–1058 (1972)
46. Pothier, G., Tanter, É., Piquer, J.: Scalable omniscient debugging. In: Proceedings of the 22nd Annual SCM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2007). ACM, New York (to appear, 2007)
47. Renggli, L., Nierstrasz, O.: Transactional memory for Smalltalk. In: Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007), pp. 207–221. ACM Digital Library (2007)
48. Robbes, R., Lanza, M.: Characterizing and understanding development sessions. In: Proceedings of ICPC 2007 (15th International Conference on Program Comprehension) (page to be published, 2007)
49. Roberts, D., Brant, J., Johnson, R.E.: A refactoring tool for Smalltalk. Theory and Practice of Object Systems (TAPOS) 3(4), 253–263 (1997)
50. Robillard, M.P., Murphy, G.C.: Feat: A tool for locating, describing, and analyzing concerns in source code. In: Proceedings of 25th International Conference on Software Engineering, pp. 822–823 (May 2003)
51. Röthlisberger, D., Denker, M., Tanter, É.: Unanticipated partial behavioral reflection: Adapting applications at runtime. Journal of Computer Languages, Systems and Structures 34(2-3), 46–65 (2008)
52. Röthlisberger, D., Greevy, O., Nierstrasz, O.: Feature driven browsing. In: Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007), pp. 79–100. ACM Digital Library (2007)

53. Röthlisberger, D., Greevy, O., Nierstrasz, O.: Exploiting runtime information in the ide. In: Proceedings of the 16th International Conference on Program Comprehension (ICPC 2008), pp. 63–72. IEEE Computer Society, Los Alamitos (2008)
54. Singer, J., Elves, R., Storey, M.-A.: Navtracks: Supporting navigation in software maintenance. In: International Conference on Software Maintenance (ICSM 2005), pp. 325–335 (September 2005)
55. Smith, B.C.: Reflection and semantics in a procedural language. Technical Report TR-272, MIT, Cambridge, MA (1982)
56. Smith, R.B., Ungar, D.: A simple and unifying approach to subjective objects. TAPOS special issue on Subjectivity in Object-Oriented Systems 2(3), 161–178 (1996)
57. Steele, G.: Growing a language. Higher-Order and Symbolic Computation 12(3), 221–236 (1999)
58. Tanter, É., Noyé, J., Caromel, D., Cointe, P.: Partial behavioral reflection: Spatial and temporal selection of reification. In: Proceedings of OOPSLA 2003, ACM SIGPLAN Notices, pp. 27–46 (November 2003)
59. Wirsing, M., Hölzl, M. (eds.): Report of the Beyond the Horizon thematic group 6 on Software Intensive Systems (2006)

On the Challenge of Engineering Socio-technical Systems

José Luiz Fiadeiro

Department of Computer Science, University of Leicester
University Road, Leicester LE1 7RH, UK
jose@mcs.le.ac.uk

Abstract. One of the main challenges raised by software-intensive systems resides in the fact that their complexity derives not so much from their size but the number and nature of the interactions that characterise their behaviour. In this paper, we discuss one of the aspects that contribute to this kind of complexity – that, more and more, people are involved, not as users, but as integral players of such systems – which requires research that can lead to new methods and techniques for engineering what are called 'socio-technical systems'.

1 Introduction

Software is becoming an integral part of a range of products and services performing vital functions in all sectors of economic and social activity. In such software-intensive systems, software applications are required to *interact*, in a seamless way, with other software components, devices, sensors, and humans. Examples include large-scale heterogeneous systems such as: embedded systems for all sorts of industry-relevant applications (automotive, avionics, etc); systems controlling critical infrastructures such as defence, energy, health, telecommunications, and transport; business applications with decision-making capabilities; and systems that ensure essential services for the functioning of society (e.g. e-government and e-learning).

One of the main challenges raised by software-intensive systems results from the fact that their complexity derives not so much from their size but the number and nature of the interactions (planned and unplanned) that characterise their behaviour. In this paper, we discuss one of the aspects that contribute to this kind of complexity: the fact that, more and more, people are involved, not as users, but as integral players of such systems.

Projects that are currently being funded and the remit of future calls already announced by several different research funding agencies address the involvement of people with ICT from a user-centric perspective. For example, this can concern adaptability of software to a user's physical or emotional state and needs, or ensuring that people's activities are supported (even enhanced) by ICT systems that are context and situation aware, inter alia. Instead, the challenge that we address in this paper concerns the 'logical' involvement of people as players in systems, contributing to the achievement of a global system goal (and not necessarily that of their own individual goals). That is to say, we are concerned with the role that people play in the services that are provided collectively by systems, and the ability of such systems to procure and bind to people at real time for the provision of required services.

M. Wirsing et al. (Eds.): Software-Intensive Systems, LNCS 5380, pp. 80–91, 2008.

One of the bottlenecks that needs to be removed for supporting the engineering of such software-intensive systems is the fact that the toolboxes that we have built for modelling and implementing complex software systems cannot cope with interactions that are not causal but only biddable (i.e., whose execution cannot be ensured by software). These toolboxes also fail to take into account the fact that people behave according to different degrees of compliance, entitlement, and normative positions relative to rôles that they are expected to play in such systems. This is the scenario that we discuss in the paper. In Section 2, we provide further motivation to the research area and its challenges. In Section 3, we identify some of the scientific and technical aspects of the challenge. In Section 4, we discuss what (and how) we can use from the state of the art. Finally, in Section 5, we present some concluding remarks.

2 Socio-technical Systems

Following a terminology that was coined 60 years ago in the area of organisational theory, we call *socio-technical systems* those software-intensive systems that involve complex interactions between software components, devices and social components (people or groups of people), not as users of the software but as players engaged in common tasks. As already explained, our particular interest in such systems is in providing the means for software components to be built that can interact with people in order to ensure that the behaviour that emerges from such interactions meets required global system properties. For instance, this means having the right methods and techniques to develop software that can control the way people interact with given mechanical devices (say a doctor operates a ventilator), but also provide the context that enables people to act according to the roles that are assigned to them within the system (say a surgeon is given access to private data of the patient and the instruments required for the operation that he/she is performing).

A key property of socio-technical systems is that they need to adapt to changes that occur in the domain in which they operate in order to ensure that they deliver the services to or control the behaviour of the components. For instance, one can think that a software component assigns different permissions to different categories of staff when controlling a given piece of equipment. Hence, in the case of health systems, if a doctor replaces, say, a nurse during an operation, the software should adapt itself to the operating conditions that correspond to the change of interactions.

A particular difficulty in engineering adaptive socio-technical systems derives from the fact that given organisational rules may need to be violated for the system to reconfigure itself to operate in what are non-normative or sub-ideal situations. For instance, under normal circumstances, the software that is controlling a routine check-up may prevent a nurse from operating some kinds of devices but, if an emergency is detected, the software should adapt to the new role that the nurse is required to perform in, say, a life-critical operation, by withdrawing some of those restrictions and providing information that a doctor would normally know or have access to (like an allergy to penicillin).

What makes socio-technical systems so distinctive and worthy of specific study is the fact that social components cannot be designed, as software and mechanical/hardware entities can, to comply with system rules; instead, they constitute what Michael Jackson calls a 'biddable domain': they can be "*enjoined to adhere to a certain behaviour, but may or may not obey the injunction*" [17]. More precisely, socio-technical systems operate in domains that, like e-health, involve both interactions that are *causal* and interactions that are only *biddable*:

- Through causal interactions, software can bring about changes required to ensure correct behaviour of the whole system like switching on a respiratory system during an operation. For instance, software can ensure that data is transmitted in the right format between the instruments that are monitoring the vital signs of a patient and those that are controlling the administration of certain drugs during an operation, and prevent a nurse from accidentally violating given safety limits; this is a case in which the interaction between the software and the nurse is "causal".
- Other interactions are not causal: for instance, software cannot control that data that needs to be monitored or controlled by staff is actually acted upon; software can control that the right data finds its way to the right reports and made available to the people with the right permissions, but not that people use it in the right way, or that people assume only the roles for which they are qualified or authorised. Such interactions, which typically involve social components (human or organisational), are only biddable; 'right' is a notion established through norms and regulations to which people or social bodies are supposed to abide but can well violate.

We must stress that biddable interactions should not be confused with unreliable causal ones. Unreliability of technical components due to the possibility of failure is a dependability concern that can be minimised by improving the technology and replacing unreliable units as new generations of components become available. Usually, there are measurable parameters that are known to vary within fixed ranges, making it possible for systems to be engineered in such a way that they can be guaranteed to operate with agreed levels of reliability. In contrast, unpredictability is intrinsic to human behaviour, which suggests that it needs to be addressed as a feature of social components and not as a fault.

A particular concern in the development of socio-technical systems is to make them able to self-adapt, at run-time, to situations in which social components are not operating according to their 'job profile', be it because they violated given rules or the context changed in ways that requires them to operate under a new role for which they are not qualified. In such circumstances, we cannot force the social components to change their behaviour, but we can reconfigure the technical and social system that they are interacting with in order to adapt it to a new operating context. Of course, people can be trained to operate with given equipment, and equipment can be designed so as to make it more human-friendly, but these are concerns that fall under the scope of *usability* – itself a challenge but not the one that concerns us here.

The following are some of the research challenges that we have identified:

- *Methodology* – One is naturally drawn to think of adopting an Artificial Intelligence methodology, but the nature of the challenge requires a different approach. Much of AI is 'agent-centric', i.e. AI approaches are typically oriented to satisfying the goals of self-interested and autonomous individuals – agents. In such multi-agent systems, the emphasis is on the ability of agents to coexist in a shared environment and pursue their respective goals in the presence and/or in co-operation with others. A methodology for socio-technical systems should be 'interaction' or 'context'-centric in the sense that the focus should be in the contexts of interaction that software should enable, not so much in the agents that would operate in such contexts. This is also different from making devices/artefacts "intelligent": it's the contexts in which such (dumb) artefacts and (potentially clever) people interact that need to be "intelligent" (for lack of a better word). It is not Artificial (agent) Intelligence that one needs but perhaps Artificial (collective) Consciousness in the sense of forms of (purposeful) awareness developed in identified contexts.
- *Formalisms and languages* – As discussed above, interactions with people are not always causal but biddable: one can ask a person to do something without being sure that they will do it. The kind of formalisms that one tends to use in software and system engineering assume causal domains, i.e. actions that one bids for will be taken (unless the machine is broken or faulty, in which case it should be replaced or repaired). The same problem can be recognised in the techniques that are typically used for modelling workflows in organisations: most of the time, they are based on causal models of interaction and fail to take into account the fact that human reactions cannot be programmed or hardwired; workflows tend to be implemented in ways that are far too coercive and rigid to sustain interactions with people, leading to fatal incompatibilities with human forms of interaction that derive from more relaxed and casual, if not opportunistic behaviour. Therefore, one would need to develop formalisms that reflect mixed causal and biddable domains.
- *Ideality* – In socio-technical contexts, one needs to be able to represent and reason about different levels of ideality, where sub-ideality is not the same thing as inconsistency but a deviation from a norm or desired situation. In particular, one the challenges that needs to be met is to be able to endow systems with the ability to self-reconfigure as a means of adapting to sub-ideal situations, possibly by minimising levels of service degradation.
- *Compositionality* – In software and system engineering one typically develops parts or components in ways that they can be put together, i.e. we address 'physiological' complexity [13]. In the kind of environments that we have in mind, people and machines/artefacts are not components in this sense: they preexist the environments that they join, they do not result from a process of decomposition. The contexts that we need to provide for them to operate together need to address social complexity [13] as it arises from their interactions, which involves unpredictability and adaptability.
- *Interference* – In the absence of compositionality, one needs to take into account the way contexts interfere when they overlap, leading potentially to multiple degrees of sub-ideality.

3 Scientific and Technical Aspects

The engineering of software-intensive systems in general, and socio-technical systems in particular, requires a methodological approach in which software components are no longer produced *ab initio* to be integrated in well-defined contexts but, instead, endowed with increasing levels of autonomy and ability to be connected, at run-time, to other system components. Although individual software components may well be designed according to classic correctness criteria, the challenge now is to make sure that the components and the interactions in place at any given state of the system make required properties emerge, making the system *fit for purpose*. This is different from designing systems that satisfy given customer requirements and, in a classical sense, are *correct*.

Therefore, there is a challenge to develop methods, tools and theoretical foundations for socio-technical systems that address fitness instead of correctness. Figure 1 recognises the three different levels at which we need to operate: the models that we build and use to control the evolution of the system; the formalisms in which we can reason about and simulate the behaviour of given configurations of a system; and the physical entities (software, devices, people) and their interactions that constitute the realisations of the system in operation.

One of the goals of the middle layer is to offer a uniform model from which one can infer properties and validate scenarios of possible configurations. However, these models should also be used to define the reconfiguration operations through which the system can (self-)adapt to changes in the domains of operation by detecting that the models no longer fit the domain. For this purpose, we need to work on the relationships between the three levels. The property layer is of a formal nature, either provided by a logic in which properties can be formulated and reasoned about, or scenarios that can be simulated and animated directly from the models. This is represented in Figure 2 through the symbol $\vdash$, which should be taken to represent not only logical inference but also simulation and animation.

The other important relationships are the ones between the models and the real domains, what in the picture we represent through dashed arrows labelled *fT, fA, fS* that we call *fits*. Depending on the nature of the domain, this fit can be formally defined and checked. This is what happens in the case of software domains, in which case the fit expresses the way a program *implements* a specification. In the case of the domains that arise in control and embedded systems, the fit needs to operate an abstraction from a model of the target plant to the mathematical domain over which the models are expressed; research in hybrid system modelling provides examples of such abstractions. In the case of other domains, like humans, the fit cannot be formalised. In this case, the model then acts as a normative design, i.e. it expresses the norms that social components are expected to observe and on the basis of which the interactions make require properties emerge. What is important is that, in case of both devices and social components, these fits can be monitored so that we can detect situations in which they cease to be valid, either because social components fail to operate according to the norms, or devices are not operating within given bounds.

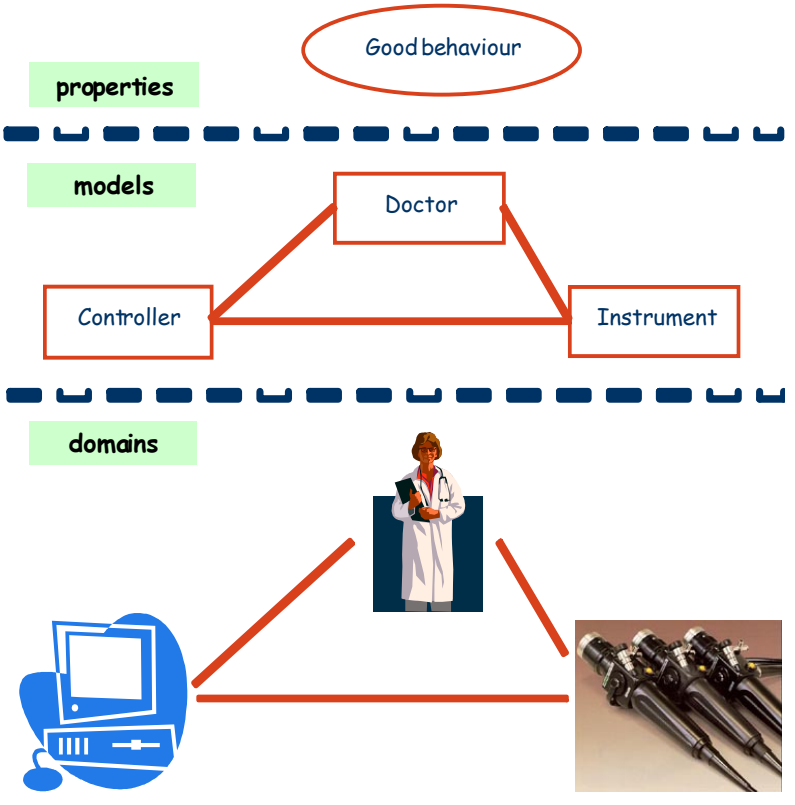


Fig. 1. 3-layered model

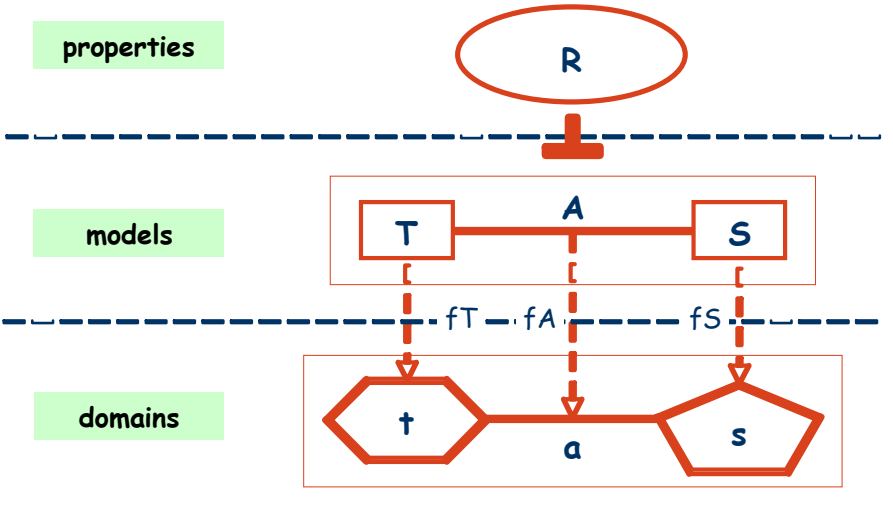


Fig. 2. The formal domains and relationships between them

Therefore, the model layer should provide the means for:

- Validating the correctness of configurations through formal verification of properties (e.g. model-checking), simulations, testing of scenarios (e.g. for possible feature interactions), and so on;
- Defining the reconfiguration operations that should be performed when given fits break for social or technical components.

Given this, we foresee several research directions that would need to be pursued:

- Scientific foundations and models that can support processes and a methodology for adaptation, including design techniques that allow one to interleave design time and run-time activities;
- Mathematical models for interconnection and emergence; it is necessary to develop a framework in which different models of system behaviour can be brought together, in order to be able to reason about emergent properties; examples include stochastic models that can reflect unpredictable behaviour, deontic logics for normative systems that can provide models for biddable domains, besides more traditional models for hybrid systems; algebraic techniques could be used for emergence and interactions;
- Social models of human components that can provide hierarchies of roles with associated notions of responsibilities, capabilities, duties, inter alia;
- Support for verification, validation, and simulation for properties such as confidentiality, security, inter alia;
- Means for testing the validity of fits, e.g. critical equipment is monitored for minimal levels of operation (through sensors), and human components are required to sign-in for the role that they play;
- Formal models of reconfiguration (language and semantics) for adaptability, from (on-line) algorithms for adaptation and reconfiguration to high-level languages and mechanisms that can express fine and coarse grained evolution;
- Engineering socio-technical systems for specific domains such as e-health, e-government, e-transport;
- Notations such as extensions of the UML.

4 Progress Beyond the State of the Art

In this section, we discuss a (necessarily limited and biased) number of the methods and techniques that we think can contribute to the engineering of socio-technical systems and point out to their limitations and ways of overcoming them.

4.1 Normative Systems

The usefulness of deontic logic for modelling the behavioural aspects of systems has been recognised by more than 20 years [e.g., 12,20,21,23,27]. Basically, deontic logic can be said to provide a formal framework for dealing with the notions of permission, prohibition, and duty or obligation [29]. The topic is also the subject of a series of specialist conferences (DEON) held biannually since 1991 [22].

What is new and appealing in these deontic accounts of behaviour from the point of view of interactions with social components is the clear separation that is achieved at the formal level between the descriptive aspects ('how things are') and the prescriptive aspects ('how things should be'). Indeed, the use of the deontic concepts permits the definition of correct or *normative* behaviour of a system, but leaves open the possibility of meeting forms of behaviour that do not comply with the norms. That is to say, this formal framework does not force us to work only within the context of normative forms of behaviour: even if normative behaviours are not possible, a deontic model is not necessarily inconsistent. Instead, it allows the derivation of information from violation states in a positive way: as pointed out in [30], information on which violations can take place can be used to decide on which corrective actions must be taken, or on which sanctions to apply when facing non-compliant behaviour [23], and so on.

The advantages of using deontic concepts for modelling organisational processes is also well established [e.g. 1,4,24]. Because an organisational model needs to capture models and enforce social patterns of behaviours of business processes operating in open environments, one needs mechanisms to systemise, defend and recommend right and wrong behaviour which in turn can inspire trust into the processes that will join them. It is essential that one can monitor both the expected (requested) and the entitled (empowered) aspects of actions, to deal with possible violations (of what needs to be done or what one is entitled to do) by detection and sanction, and so on, for which it is necessary to have a formal framework over which one can reason about different degrees of compliance, entitlement, and normative positions in general [19,24].

This is why we consider that deontic logic and related formalisms that support normative systems provide a good starting point for modelling the biddable aspects of interactions, much in the same way that other modal logics like temporal logic have been used for modelling their causal aspects.

4.2 Architectural Connectors

The area of software architecture [e.g.,10,11,25] has been promoting an interaction-centric approach to modularisation through connectors [2] that coordinate interactions as external entities. In the connector-based approach, coordination mechanisms can be superposed dynamically over the interactions without the components being aware of the way their interactions are being coordinated. In this approach, evolution can be made to be compositional over the architectural structure of the system [3], and self-adaptation can be addressed at the architectural level [15].

Architectural approaches have proved to work well when the interactions being coordinated are among software components in traditional software configurations. An architectural connector consists of (1) a set of roles that capture the types of the components that can be interconnected and (2) a glue that enforces an interaction protocol between any components that instantiate the roles. The formalisms that are used for the roles and glue tend to be process-oriented, for instance CSP processes [16] in the language Wright [2], action-based reactive systems in CommUnity [14], and event-condition-action rules in the CCC approach [3]. As already argued, these formalisms are not adequate for interactions that involve social components. Hence, one would need to investigate how architectural connectors can be extended to socio-technical systems:

- If one considers the notion of architectural connector in Wright [2], CommUnity [14], or CCC [13], extensions of the notion of role would need to be developed that can model human roles and capabilities, and also formalisms through which the glue (the software executed by the connector) can take into account the biddable nature of interactions. This is where graph-based representations of roles and capabilities and deontic logics would provide a good starting point.
- On the other hand, socio-technical systems require connectors to be dynamic in the sense that they will be able to reconfigure themselves in order to adapt to changes of context. These reconfigurations may imply changing components but also just the roles that they play and the policies (glue) according to which they interact.

4.3 Problem Frames

The interaction-centric view of architectures blends well with Jackson's Frames approach to problem analysis and decomposition as explained in [5]. Architectural techniques lie essentially in the 'solution domain', i.e. they address the configuration of a system whose components have been identified, and its evolution through reconfiguration with identified new components and/or connectors. However, for such an architectural framework to provide effective support for modelling socio-technical systems, one needs to address the 'problem domain'. By this we mean the ability to work over an explicit decomposition of the problem domain that can identify the components that intervene in the system, the assumptions that they make about each other in the way they are interconnected, and the way the satisfaction of given system requirements emerges from these interactions. This is exactly the role of approaches to problem analysis and decomposition such as Jackson's Problem Frames [18].

Problem Frames encapsulate both real world and system objects, and describe the interactions between them. People, as system objects, lie in biddable domains which are different from causal domains in that correct behaviour cannot be ensured by causing objects to act in pre-determined ways. The problem decomposition and architectural techniques developed by Hall and Rapanotti [26] to incorporate Problem Frames in representing human knowledge and guiding development of real-world socio-technical systems seem to provide a good starting point, in particular in what concerns the formalisation of collaboration patterns that have been proposed for modelling both human-human and human-machine interactions [8].

Like for the 'architectural connector' view, a major difference between Problem Frames and what we require for socio-technical systems is in the dynamic nature of the requirements, i.e. when one needs to adapt the interactions to changes of context. Typically, where a problem frame would capture a normative or ideal behaviour that enforces a requirement, we need some sort of scheme that can also capture sub-ideal situations that may arise from, say, humans violating permissions or not obeying injunctions, or changes of context that violate the roles assigned to social components.

4.4 Program Correctness

The aim of the models and the mathematical techniques that we see supporting the engineering of socio-technical systems is *not* to establish correctness as normally

understood in program development and software engineering. Our purpose is to make sure that the components and the interactions in place at any given state of the system make required properties emerge, making the system *fit for purpose*. This property relies on the relationships between the models and the real domains, what we call *fits* (recall Figure 2). Depending on the nature of the domain, this fit can be formally defined and checked. This is what happens in the case of software domains, in which case the fit expresses the way a program *implements* a specification.

What we said holds in an ideal situation. In a sub-ideal situation, not all requirements of a purpose can be satisfied and it is important to measure the degree of fitness of the different possible behaviours in order to find the "best fit". Possible approaches to measuring fitness could be to build on results on test coverage (for a comprehensive overview see [6]), topological approximations or soft constraints [7]. The testing approach would try to identify inputs with a high probability of incorrect behaviour and to extrapolate from these test cases. In topological approximations, initial segments of program traces are compared. With soft constraints logical properties of the program are declaratively specified and fitness is computed by checking constraint satisfaction over an appropriate semi-ring. We deem soft constraints to be a promising approach to fitness as they have proved to be very flexible in describing non-functional requirements [9], partial constraint satisfaction [7], and preferences [31].

In the case of the domains that arise in control and embedded systems, the fit needs to operate an abstraction from a model of the target plant to the mathematical domain over which the models are expressed; research in hybrid system modelling provides examples of such abstractions. In the case of social domains, the fit cannot be formalised; the model acts as a normative design, i.e. it expresses the norms that social components are expected to observe and on the basis of which the interactions make require properties emerge.

From the point of view of architectural connectors, it is important that these fits are monitored so that we can detect situations in which they cease to be valid, either because social components fail to operate according to the norms, or devices are not operating within given bounds. The latter is a typical situation of dependability for which several techniques have been developed; the former is essential for ensuring dynamicity of trust in the sense of socio-technical systems.

5 Concluding Remarks

Socio-technical systems raise additional challenges over the engineering of software-intensive systems because they exhibit interactions that involve social components, which cannot be modelled and controlled in the same way as those that involve only technical components (software or otherwise). We have argued that the formalisms, methods, languages, and techniques that are normally used for supporting the engineering of software systems cannot be applied directly to socio-technical systems. We have also pointed to ways in which we think the state of the art can be extended to meet the new challenges.

In particular, we foresee that software engineering methods and techniques such as problem frames and architectural connectors can be extended in several ways, including the use of different logics and mathematical domains (e.g. deontic logic).

However, these extensions need to be taken with care because the intended scope is *not* of traditional software engineering, but socio-technical system engineering. That is, our interest in problem frames and architectural connectors is not for supporting software requirements, specification and design, but in the methods and techniques that we can borrow for supporting the engineering of systems that may involve software applications as technical components but, primarily, complex interactions between social and technical components. In particular, the models of the social and technical components need only to be indicative in the sense of [18], i.e. assumptions not requirements. Naturally, assumptions on technical components may well need to be guaranteed by programs that control them, but the goal of the models layer in Figure 1 is not to identify the requirements that such programs need to fulfil: we are do not see scope for a classical top-down method of engineering socio-technical systems but a method of developing software applications that can make socio-technical systems adaptable and 'fit for purpose'.

Acknowledgments

We would like to thank Maurice ter Beek, Schahram Dustdar, Dimitrios Georgakopoulos, Stefania Gnesi, Reiko Heckel, Birna van Riemsdijk, Alessandra Russo, and Martin Wirsing for the many discussions that we had on these topics. We would also like to thank all the participants of the BTH meetings with whom we had the privilege of discussing these research challenges.

References

1. Abrahams, A., Eyers, D., Bacon, J.: An asynchronous rule-based approach for business process automation using obligations. In: Proc. 2002 ACM SIGPLAN workshop on Rule-based programming, pp. 93–103. ACM Press, New York (2002)
2. Allen, R., Garlan, D.: A Formal basis for architectural connectors. ACM TOSEM 6(3), 213–249 (1997)
3. Andrade, L.F., Fiadeiro, J.L.: Architecture-based evolution of software systems. In: Bernardo, M., Inverardi, P. (eds.) SFM 2003. LNCS, vol. 2804, pp. 148–181. Springer, Heidelberg (2003)
4. Barbuceanu, M., Gray, T., Mankovski, S.: Coordinating with obligations. In: Proceedings of the second international conference on autonomous agents, pp. 62–69. ACM Press, New York (1998)
5. Barroca, L., Fiadeiro, J.L., Jackson, M., Laney, R., Nuseibeh, B.: Evolving problem frames: a case for coordination. In: de Nicola, R., Meredith, G. (eds.) COORDINATION 2004. LNCS, vol. 2949, pp. 5–19. Springer, Heidelberg (2004)
6. Binder, R.V.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley, Reading (1999)
7. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint satisfaction and optimization. Journal of the ACM (JACM) 44(2), 201–236 (1997)
8. Bolloju, N.: Improving the quality of business object models using collaboration patterns. In: CACM, vol. 47, pp. 81–86 (2004)
9. De Nicola, R., Ferrari, G., Montanari, U., Pugliese, R., Tuosto, E.: A basic calculus for modelling service level agreements. In: Jacquet, J.-M., Picco, G.P. (eds.) COORDINATION 2005. LNCS, vol. 3454, pp. 33–48. Springer, Heidelberg (2005)

10. Dustdar, S.: Caramba – a process-aware collaboration system supporting ad-hoc and collaborative processes in virtual teams. In: Distributed and Parallel Databases, vol. 15(1), pp. 45–66 (2004)
11. Dustdar, S., Gall, H.: Architectural concerns in distributed and mobile collaborative systems. Journal of Systems Architecture 49(10-11), 457–473 (2003)
12. Esteva, M., Padget, J., Sierra, C.: Formalizing a language for institutions and norms. In: Meyer, J.-J., Tambe, M. (eds.) ATAL 2001. LNCS (LNAI), vol. 2333, pp. 348–366. Springer, Heidelberg (2002)
13. Fiadeiro, J.L.: Modelling for software's social complexity. IEEE Computer 25(1), 34–39 (2007)
14. Fiadeiro, J.L., Lopes, A., Wermelinger, M.: A mathematical semantics for architectural connectors. In: Backhouse, R., Gibbons, J. (eds.) Generic Programming. LNCS, vol. 2793, pp. 190–234. Springer, Heidelberg (2003)
15. Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., Steenkiste, P.: Rainbow: architecture based self adaptation with reusable infrastructure. IEEE Computer 37(10), 46–54 (2004)
16. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall International Series in Computer Science (1985)
17. Jackson, M.: Software Requirements and Specifications: A lexicon of practice, principles and prejudices. Addison-Wesley, Reading (1995)
18. Jackson, M.: Problem Frames: Analysing and Structuring Software Development Problems. Addison-Wesley, Reading (2000)
19. Jones, A., Sergot, M.: On the charactertisation of law and computer systems: the normative systems perspective. In: Meyer, J.J., Wieringa, R. (eds.) Deontic Logic in Computer Science. Normative System Specification. Wiley, Chichester (1993)
20. Lomuscio, A., Sergot, M.J.: Deontic interpreted systems. Studia Logica 75(1), 63–92 (2003)
21. McCarty, L.: Permissions and obligations. In: IJCAI 1983, pp. 287–294 (1983)
22. Meyer, J.-J., Wieringa, R. (eds.): Deontic Logic in Computer Science: Normative System Specification. John Wiley & Sons, Chichester (1993)
23. Minsky, M., Lockman, A.: Ensuring integrity by adding obligations to privileges. In: Proc. 8th IEEE Int. Conf. on Software Engineering, pp. 92–102 (1985)
24. Padmanabhan, V., Governatori, G., Sadiq, S., Colomb, R., Rotolo, A.: Process modelling: the deontic way. In: Proceedings of the 3rd Asia-Pacific conference on Conceptual modelling, vol. 53, pp. 75–84 (2006)
25. Perry, D., Wolf, A.: Foundations for the study of software architectures. ACM SIGSOFT Software Engineering Notes 17(4), 40–52 (1992)
26. Rapanotti, L., Hall, J., Nuseibeh, B., Jackson, M.: Architecture-driven problem decomposition. In: Proc. 12th IEEE International Requirements Engineering Conference (RE 2004), pp. 80–89. IEEE Press, Los Alamitos (2004)
27. Sergot, M.: Normative positions. In: MacNamara, P., Prakken, H. (eds.) New studies in deontic logic and computer science, pp. 289–308. IOS Press, Amsterdam (1998)
28. Sergot, M.J.: Modelling unreliable and untrustworthy agent behaviour. In: Dunin-Keplicz, B., Jankowski, A., Skowron, A., Szczuka, M. (eds.) Monitoring, Security, and Rescue Techniques in Multiagent Systems, pp. 161–178. Springer, Heidelberg (2005)
29. von Wright, G.: Norm and Action. Routledge and Kegan Paul (1963)
30. Wieringa, R., Meyer, J., Weigand, H.: Specifying dynamic and deontic integrity constraints. In: Data and Knowledge Engineering, vol. 4, pp. 157–189 (1989)
31. Wirsing, M., et al.: Semantic-based development of service-oriented systems. In: Najn, E., et al. (ed.) FORTE 2006. LNCS, vol. 4229, pp. 24–45. Springer, Heidelberg (2006)

Design of Complex Cyber Physical Systems with Formalized Architectural Patterns

Lui Sha and José Meseguer

University of Illinois at Urbana-Champaign
{lrs,mesequer}@cs.uiuc.edu

Abstract. The design of cyber physical systems (CPS) presents many challenges because of their complexity, strong safety requirements, distribution, and real-time nature. We propose a novel paradigm, based on the idea of using simplicity to control complexity, to achieve highly reliable CPS designs. The goal is to embody design rules of this complexity-control nature in highly reusable, very robust, and formally verified architectural patterns. We discuss some preliminary work and experiments illustrating how this can be done for CPS systems.

1 Introduction

The convergence of sensing, control, communication and coordination in cyber-physical systems (CPS) such as modern airplanes, the power grid, transportation systems, and medical device networks, poses enormous challenges because of their complexity. The following aspects seem particularly challenging:

- Distributed and concurrent interactions between the cyber and physical subsystems.
- Strong QoS requirements including real-time, fault tolerance, safety and security requirements.
- The risk of dramatic losses, including losses of human lives, when these systems malfunction and/or are penetrated by cyber–terrorist attacks.

System complexity implies an overwhelming number of states to be checked. This is one of the greatest challenges to the development of reliable software. Unfortunately, many useful features are inherently complex. Furthermore, in many complex CPS systems, there are components that are beyond verification. For example, after major surgery, a patient is allowed to "operate" an infusion pump with potentially lethal pain killers (patient controlled analgesia (PCA)). When pain is severe, the patient can push a button to get more pain-relieving medication. This is an example of a safety critical device controlled by an error-prone operator (the patient). Nevertheless, the PCA system as a whole needs to be certifiably safe in spite of mistakes made by the patient. In the following, we first consider software component reliability.

There are two main approaches to software reliability. One is the fault avoidance method using formal specification and verification methods and a rigorous software development process such as FAA's DO 178B standard for flight control software. Another approach is based on using methods that make systems fault-tolerant by

M. Wirsing et al. (Eds.): Software-Intensive Systems, LNCS 5380, pp. 92–100, 2008.

design, for example by design diversity [17]. A promising hybrid approach, advocated in [1] and also in recent work on "runtime verification and monitoring" (see, e.g., [7] and references there), consists in developing fault-tolerant system architectures that can detect system faults and can use highly reliable, verified components to ensure safe behavior even in the presence of faults and of unreliable components. Fault-avoidance methods, such as methods based on model checking or theorem proving, provide high confidence, but are hard to scale up when used in isolation as the only approach to achieving system reliability. The trend towards large Cyber Physical Systems (CPS) makes the application of fault avoidance methods even more difficult. On the other hand, methods based on fault-tolerance that do not leverage formal methods technology make systems more robust, but cannot by themselves provide high assurance. We believe that a hybrid approach that combines fault-tolerant architectures with formal verification is needed to support the design of safe and highly robust CPS systems. In Section 2 we first review the results from [1], which examine the relation between software diversity, complexity and software reliability. Building upon this insight, we outline a method based on formalized architecture patterns as a solution for building safe and reliable CPS systems in Section 3. In Section 4 we briefly discuss some related work and present some conclusions.

2 Diversity, Complexity and Software Reliability

Will we have a more reliable software system by putting all the efforts into a single program, or by dividing the limited resources to develop several programs for the same purpose that provide diversity and can be used for fault-tolerance? To get some insight into this question, let us develop a simple model to analyze the relationship between reliability, development effort, and the logical complexity of software. Computational complexity measures the amount of resources needed to complete a computation. In a similar way, the "logical complexity" of a software system should measure the amount of effort needed to verify its correctness.

Methods that can decrease a system's logical complexity are of paramount importance. It is important to note the difference between initial logical complexity and residual logical complexity. A program could have high logical complexity initially. However, if it has been formally verified and can be used as is[1], then its residual logical complexity is zero. The value of formally verified architectures is to make residual complexity small. In the following, the term "complexity" refers to residual logical complexity unless stated otherwise. Based on what has been observed in software development, we propose three postulates:

- *P1: Complexity Breeds Bugs:* Everything else being equal, the more complex the software project is, the harder to make the resulting system reliable.
- *P2: All Bugs Are Not Equal*: System design errors are much more important and costly than coding errors; they are also subtler and more difficult to detect and correct.
- *P3: All Budgets are Finite:* There is only a finite amount of effort (budget) that we can spend on any project.

[1] It is important to point out that we cannot simply use a known reliable component in a new environment without verifying the assumptions made by the component.

P1 implies that for a given mission duration t, the reliability of software decreases as complexity increases. P2 implies that for a given degree of complexity, the reliability function has a monotonically decreasing rate of improvement with respect to development effort. This is because easy to spot coding errors will be found and fixed with modest effort, but subtle coding errors and, more importantly, design errors are much more costly to find and fix. P3 implies that diversity is not free. That is, if we go for diversity, we must divide the available effort in some way.

For example, a simple reliability model that satisfies the three postulates is the commonly used exponential reliability function $R(t) = e^{-\lambda t}$ and assume that the failure rate, λ, is proportional to the software complexity, C, and inversely proportional to the development effort, E. That is, $R(t) = e^{-kCt/E}$. To focus on the interplay between complexity and development effort, we normalize the mission duration t to 1 and let the scaling constant $k = 1$. As a result, we can rewrite the reliability function with a normalized mission duration in the form $R(E, C) = e^{-C/E}$. Under this model, the higher the complexity, the more effort is needed to achieve a given degree of reliability. $R(E, C)$ also has a monotonically decreasing rate of reliability improvement, demonstrating that the remaining errors are subtler and, therefore, detecting and correcting them requires more effort. Finally, the available budget E should be the same for whatever fault-tolerance method is used.

Under this model, we were able to show that single version programming has higher reliability than 3 version programming under a wide range of conditions [1]. However, the highest reliability architecture structure is to have a simple and verifiable alternative that can provide the essential services in spite of the faults and failures in the complex full feature alternative. To illustrate this idea, let us consider the problem of sorting. In sorting, the critical property is to sort items correctly. The desirable property is to sort them fast. Suppose that we could formally verify a Bubble Sort program but were unable to verify a ComplexFastSort program. Can we make use of this slow Bubble Sort as a watchdog for ComplexFastSort? Yes, we can.

As illustrated in Figure 1, to guard against all possible faults of ComplexFastSort, we put these two programs in two virtual machines. In addition, we develop a verified object called "permute" that will: (i) allow ComplexFastSort to perform all the list operations to rearrange the order of the input item in the input list, but *not* to modify, add or delete any list item; and (ii) check in linear time that the output of ComplexFastSort is indeed sorted. Finally, we set a timer based on the promised speed of ComplexFastSort if it is supposed to be faster than the BubbleSort. If ComplexFastSort does finish in time and we check that the answer is correct, then the result is given as output; if it does not finish in time or does so but with an incorrect answer, then BubbleSort sorts the data items.

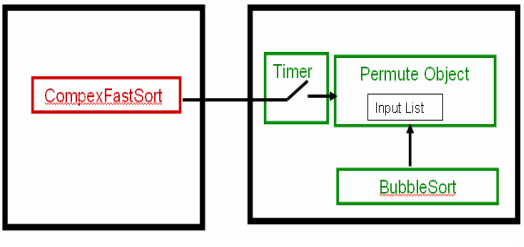


Fig. 1. An Always Correct Sorting System

It is important to note that this sorting system is provably correct for all possible new sorting components, including the use of a human to do the sorting. Furthermore, there is a lower bound on performance and this lower bound can be improved by

replacing BubbleSort by a faster and formally verified sorting program. The moral of this story is that we can exploit the features and performance of complex components even if we cannot verify them, as long as we can guarantee the critical properties by simple software and an appropriate architecture pattern. In this way we can leverage the power of formal methods to provide high assurance in the system development process. We call this architecture principle "using simplicity to control complexity."

Checking the correctness of an output before using it, such as in the sorting example, belongs to a fault tolerant approach known as recovery block [19]. However, in CPS applications, it is *not* always possible to determine if every command from a complex controller is correct (meeting the specifications). Fortunately, it is safe to execute an incorrect control command, provided that the plant's stability margins can still be met and hence the resulting errors are recoverable [1]. The simplex architecture allows us to safely exploit complex high performance control subsystems that may have residual errors by using a simple high assurance subsystem and by monitoring the resulting stability margin if a command were executed (Figure 2) [1]. That is, if a command might lead to instability, we always reject it; otherwise we give it the benefit of doubt. A noteworthy example of "using simplicity to control complexity" in practice is the flight control system of Boeing 777 [18]. It uses triple-triple redundancy for hardware reliability. At the software application level, it uses two controllers. The sophisticated control software specifically developed for Boeing 777 is the normal controller. The secondary controller is based on the control laws originally developed for Boeing 747. The normal controller is much more complex and is able to deliver optimized flight control over a wide range of conditions. On the other hand, control laws developed for Boeing 747 have been used for over 25 years. It is a mature old technology – simple, reliable and well understood. From our perspective, we will call it a simple component, since it has low residual complexity. To exploit the advantage of advanced control technologies and to ensure a very high degree of reliability, Boeing 777 under the normal controller should fly within the stability envelope of its secondary controller. This is a fine example of using simplicity to control complexity.

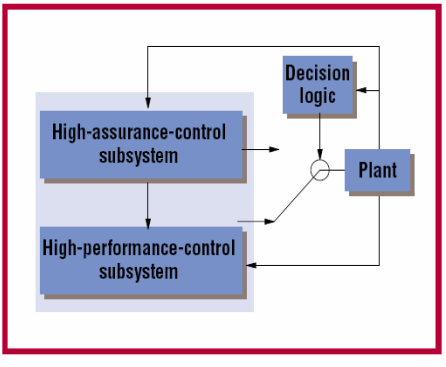


Fig. 2. Simplex architecture

In the following, we will illustrate the idea of complexity control in the context of a medical system and how we can formalize useful complexity control architecture patterns.

3 Formalized Architecture Patterns with Computer-Aided Verification

Complex and unverifiable components, e.g., human operators and highly complex software components, are unavoidable. Fortunately, we can ensure critical properties

and lower bounds on performance using: (1) formally verified complexity control architecture patterns, and (2) formally verified simple components for essential services. That is, under a given fault model we need to verify the following properties:

- Protection: The architecture software and the simple component cannot be corrupted by faults from the unverified complex software components.
- Timeliness: The simple and verified components must be executed within timing constraints.
- Fault tolerance: in spite of all the faults under the fault model, the simple, verified component will function correctly.

Since architecture patterns often need to be adapted for new application requirements, we need to not only verify a collection of commonly used architecture patterns, but also provide computer aided verification for the adaptation of architecture patterns. Furthermore, since in software practice model-based approaches are the most common way of capturing architectural designs and architectural patterns, it is important to provide formal verification support for architecture patterns expressed in software modeling languages. To make all this possible we, together with our students at UIUC and in collaboration with Artur Boronat at the University of Leicester and Peter Olveczky at the University of Oslo, Darren Cofer and Steve Miller of Rockwell Collins, and Peter Feiler, Jorgen Hansson and Dionisio de Niz of SEI are currently working on several mutually-reinforcing tasks:

- Complexity control architectures and design rules for avionics and medical systems.
- Formalized SAE AADL [3] subset to specify these architectures.
- Use of MOMENT2 [6][8] to automatically transform AADL models into algebraic expressions in Maude for formal analysis purposes and for further transformation into Real-Time Maude [9] specifications.
- Formal semantics of AADL in Real-Time Maude, and automatic transformation of AADL models into Real-Time Maude specifications based on such semantics, to provide both symbolic simulation and formal verification by model checking for AADL models.

In the following, we use a simple example to illustrate how this approach works.

Example: According to the APSF Newsletter from Winter 2004, "*A 32-year-old woman was having a laparoscopic cholecystectomy (surgical removal of the gallbladder) performed under general anesthesia. During that procedure and at the surgeon's request, a plain film x-ray was shot during a cholangiogram. The anesthesiologist stopped the ventilator for the x-ray. The x-ray technician was unable to remove the film because of its position beneath the table. The anesthesiologist attempted to help the technician, but found it*

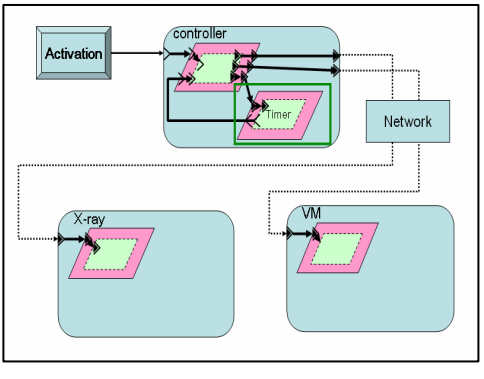


Fig. 3. Composition 1 of ventilator machine (VM) with X-ray machine and controller

difficult because the gears on the table had jammed. Finally, the x-ray was removed, and the surgical procedure recommenced. At some point, the anesthesiologist glanced at the EKG and noticed severe bradycardia. He realized he had never restarted the ventilator. This patient ultimately died."

This accident could have been prevented by automation. There are two candidate compositions, however.

Composition 1: Network the x-ray machine and ventilator together with a control station. The control station could command the ventilation to pause, the x-ray machine to take a picture, and then command the ventilator to resume. In addition, two watch-dog timers could be added to the control station. The first one limits the maximum duration of each pause. The second one ensures that pauses are separated by a minimum duration. Both of them are configuration time constants set by medical personnel. However, such a design is unacceptable, because if either the network or the control station fail, after commanding the ventilator to pause, the ventilator will be stuck at the pause state.

Composition 2: A better design is to put these two timers inside the ventilator. From an architecture perspective, the latter design minimizes the safety dependency tree into a single node: the ventilator. Under this design, as long as the ventilator is verifiably safe, the overall system is safe in spite of the faults and failures in the network, the command station and the x-ray machine. From a safety perspective, we can now safely integrate the ventilator into different networks with different but interoperable consoles and x-ray machines without recertifying the safety of the system, because the network, x-ray machine and console are not part of the safety dependency tree.

From the perspective of Simplex architecture [1], in composition 2 the ventilator is required to be verifiably safe. Once this is done, it can safely collaborate with non-safety critical devices such as the network and a command

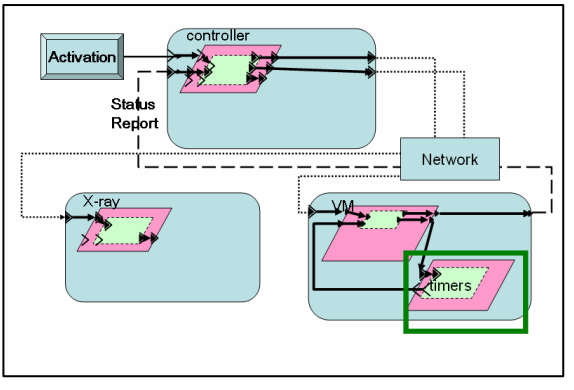


Fig. 4. Composition 2 of ventilator machine (VM) with X-ray machine and controller

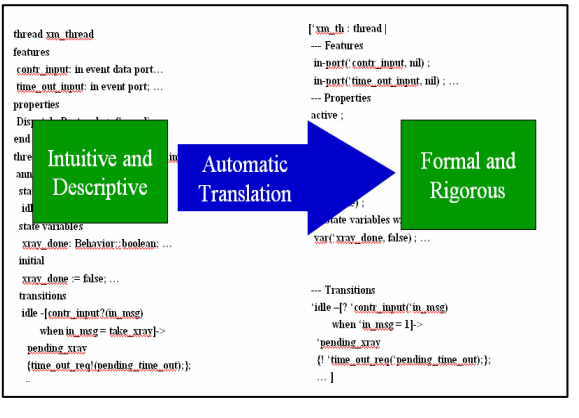


Fig. 5. AADL to Real-Time Maude Translation

station. The command station and network should be industrial grade, not certifiably safe because certifying the OS and the network is prohibitively expensive. Furthermore, if they were certified, any change in OS and network would trigger recertification. And any non-safety critical device or network information flows connected with this certified network would trigger recertification. Minimizing the use of certifiably safe components, especially the infrastructure components such as OS and network, is critical to the economics of medical device networks.

Under the Simplex architecture, non-safety critical devices can be added, modified and replaced without jeopardizing safety invariance, provided that architecture design rules are followed. This is done by ensuring that the safety invariants are satisfied by the set of safety critical components. In this example, the safety invariants of the ventilator are the limit on the maximal duration of each pause, and the limit on the minimal duration of separation between pauses. These invariants are specified by means of configuration time constant set by medical personnel and enforced by the two timers at runtime. Under the assumption that medical personnel set the constants correctly and the timers embedded in the ventilator design work, the ventilator is safe for all possible inputs from the command station, because the timeouts are not a function of inputs from the commands.

The ventilator pause is instantiated from the command station and the command goes through the network. Thus, we say that the architecture *uses* the network, x-ray and command station, but the safety does *NOT depend* on them. This "use but not depend" is a key principle of the Simplex architecture, which minimizes the use of safety critical components, while maximizing the safe utilization of non-critical components. When critical components use but do not depend on less critical components, we say that the system safety dependency relation is *well-formed*. Otherwise, we say that there is (safety) *dependency inversion*.

As illustrated in Figure 5, to check if a candidate composition is well formed is done by first developing a model of the composition in AADL with a behavior specification. The AAD model is then translated into Real-Time Maude using its rewriting logic semantics and MOMENT2. The fault model is a specification of possible incorrect state transitions. Using the Real-Time Maude models of faulty transitions in unverified components and of the system, we were able to verify by model checking that the AADL model of the ventilator operation satisfies the two safety invariants on maximum pause time and on minimum time between pauses and is therefore verifiably safe for such invariants. Furthermore, the liveness property that the X-ray will be taken during the pause of the ventilator in the absence of faults was also verified.

4 Related Work and Conclusions

Due to both space limitations and the wide range of related areas, we cannot give a proper comparison with related work. We can, however, mention some sample instances of work that is somehow related to some aspects of our approach. First of all, in the area of formal analysis tools for real-time systems we can mention other tools such as, for example, Upaal [10], HyTech [11], and Kronos [12]. As already mentioned, the approach in [1] is related to recent work in runtime monitoring and runtime verification; this is a very active area with regular conferences and it is hardly

possible to give comprehensive references; we refer to [7] and references there as a good entry point into the subject. The ideas on formally-based architectural patterns are related to work on modular system specification and verification, particularly to work supporting assume-guarantee reasoning about modular components. Again, this is a large research area, but we can mention, for example, [13][14] and references there. There is also extensive work in combining model-based approaches and formal methods that we cannot do justice to; let us just mention [15], as an alternative formal framework for MOF different from that of MOMENT2, and the work in [16], as examples of papers in that area that are close to our work. Fault tolerant approaches for engineering robust software systems are also a vast area. We can however mention [17], which provides a good survey of this area.

The convergence of sensing, control, communication and coordination in cyber-physical systems such as modern airplanes, power grid, transportation systems, and medical device networks poses enormous challenges because of their complexity. Work in all the areas mentioned above is certainly relevant and useful. However, to address the hard challenges of CPS system design, we focus on a synergistic combination of specific technologies to support model-based design of highly reliable CPS systems. These combined technologies include: architectural patterns, fault-tolerant techniques, model-based software engineering, and object-based formal specification and verification of real-time systems.

Acknowledgments. We thank all our collaborators in developing these ideas and case studies, including Artur Boronat, Darren Cofer, Peter Feiler, Steve Miller, Peter Ölveczky, and our students Tanya Crenshaw, Joe Hendrix, Minyoug Nam, Xiaokang Quiu, and Mu Sun. We also thank our sponsors, including Rockwell Collins, NSF and ONR.

References

[1] Sha, L.: Using Simplicity to Control Complexity. IEEE Software (July/August 2001)
[2] Lyu, M.R. (ed.): Fault Tolerance. John Wiley & Sons, Chichester (1995)
[3] AADL, http://www.sei.cmu.edu/products/courses/p52.html
[4] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
[5] MOMENT2, http://www.cs.le.ac.uk/people/aboronat/tools/moment2-gt/
[6] Rosu, G., Havelund, K.: Rewriting-Based Techniqes for Runtime Verification. Automated Software Engineering 12, 151–197 (2005)
[7] Boronat, A., Meseguer, J.: An Algebraic Semantics for MOF. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 377–391. Springer, Heidelberg (2008)
[8] Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation 20(1-2), 161–196 (2007)
[9] Behrmann, G., David, A., Larsen, K.G.: A Tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)

[10] Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: HYTECH: A Model Checker for Hybrid Systems. Softw. Tools Technol. Trans. 1, 110–122 (1997)
[11] Yovine, S., Kronos: A Verification Tool for Real-Time Systems. Softw. Tools Technol. Trans. 1, 123–133 (1997)
[12] Misra, J.: A Discipline of Multiprogramming. Springer, Heidelberg (2001)
[13] Viswanathan, M., Viswanathan, R.: Foundations for Circular Compositional Reasoning. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 835–847. Springer, Heidelberg (2001)
[14] Poernomo, I.: The meta-object facility typed. In: Proc. SAC, pp. 1845–1849. ACM, New York (2006)
[15] Romero, J.R., Rivera, J.E., Duran, F., Vallecillo, A.: Formal and Tool Support for Model Driven Engineering with Maude. Journal of Object Technology 6(9) (2007)
[16] Lyu, M.: Software Fault Tolerance, `http://www.cse.cuhk.edu.hk/~lyu/book/sft/index.html`
[17] Yeh, Y.C.: Dependability of the 777 Primary Flight Control System. In: Proc. Dependable Computing for Critical Applications. IEEE CS Press, Los Alamitos (1995)
[18] Tyrrell, A.M., Tyrrell, A.M.: Recovery Blocks and Algorithm-Based Fault Tolerance. In: Proceedings of the 22nd EUROMICRO Conference

Cyber-Physical Systems and Events

Carolyn Talcott

SRI International
Menlo Park, CA 94025-3493, USA
clt@cs.stanford.edu

Abstract. This paper discusses *event-based semantics* in the context of the emerging concept of Cyber Physical Systems and describes two related formal models concerning policy-based coordination and Interactive Agents.

1 Introduction

Cyber-physical systems (CPSs) integrate computing and communication with monitoring and/or control of entities in the physical world. Sensing and manipulation of the physical world occurs locally, system behavior emerges as a result of communication and other forms of interaction. Example CPSs include automobiles, aircraft, air traffic control, power grids, oil refineries, medical devices, patient monitoring, and smart structures. Software is becoming an increasingly important element of the operation of these systems, and must do so dependably, safely, securely, efficiently and in real-time.

CPSs go beyond traditional embedded and distributed systems. They are often long lasting, with 24x7 operation and must evolve without losing stability. Some CPSs or their components have stringent QoS requirements, others are more flexible. Traditional embedded and critical systems are closed, not only in the sense of closed physical locations or dedicated networks, but also closed with respect to their computational boundaries, i.e., all the participating elements in the systems are known initially. Thanks to network technology and mobility, today's embedded systems are shifting towards openness and federation This leads to multi-scale, wide area critical systems with real-time requirements, all this still with certification requirements, a major verification challenge. The openness brings more convenience and flexibility for controlling the systems. However, it also introduces extra complexity into the systems: large scale, uncertainty, and dynamics—entities can come into or leave from the systems, and makes the coordination of entities even harder.

QoS requirements, such as timing properties, fault-tolerance, security, etc., dictate how individual entities of the system being considered coordinate with each other. For instance, a deadline constraint on a task indicates that there must exist another entity that is coordinated with the constrained task. Or the deadline misses its meaning. If we consider computation is to achieve the system's functional requirements, QoS requirements are reflected through coordination among computational entities. With this view, embedded or critical systems are compositions of two main elements—computation and coordination.

A solid semantic foundation is crucial for design, deployment, monitoring and adaptation of CPSs. Such a foundation must support reasoning locally about individual components and globally about system wide properties. Our hypothesis is that *event-based*

M. Wirsing et al. (Eds.): Software-Intensive Systems, LNCS 5380, pp. 101–115, 2008.

semantics can provide such a foundation. In Section 2 we discuss the general idea of event-based semantics. A sampling of related work is presented in Section 3. This is followed by brief overviews of two specific ideas: PAGODA—a policy and goal-based approach to modeling autonomous system components, and Interactive agents—which combines a policy based coordination model and an extension of the notion of actors to include interactions other than messages.

2 Event-Based Semantics

We begin with a discussion of various notions of event, followed by essential features, and challenges to be met in developing event-based semantics for CPSs.

2.1 What Is an Event?

An instance of an action? An occurrence in time and space? A change of condition? There are many notions of event, different notions being useful for different purposes, including:

- In the actor model of computation an event is a message send or receive.
- In the process algebra model of computation an event is an action shared by two processes.
- In the world of linguistics events may happen over time or be nested. For example: Tim and Ben played World of WarCraft. They completed a quest.

We can classify event models along several dimensions:

- punctual (for example, message send/receive) vs durative (for example, filling the tank, attending a class)
- single vs stream (for example, card reader readings, periodic chemical sensor readings, or object tracking)
- change vs action/observation.

In addition, event models may have different underlying temporal models: causal ordering (before/after), discrete time, continuous time.

It is important for an event-based semantic to include many different notions of event, and to allow moving from one to another in meaningful ways.

2.2 Why Event-Based?

What are the essential features of event-based semantics? One key feature of events is that they concern interactions between components and observations rather that internal state. This enables specification and reasoning at higher-levels while integrating easily with more detailed information. Another key feature of events is the notion of causal partial order that reflects the physical reality that for events separated in space we may not be able to decide a linear order (and should not depend on it).

- Events are a natural way to think about reactive systems, such as CPSs. They provide a natural way to specify components of open systems in terms of interfaces and observable behavior. Events also can form the basis for specifying coordination and composition of components.
- Global state and local state can be abstracted to event partial orders. The causal partial order of an event semantics captures dependencies/consequences and allows reasoning about what must have happened in the past, given some reasonable assumptions about the behavior of system components. Further, by (logically) locating events, reasoning can be localized, and will scale.
- Event models can be used to give semantics to specifications, as well and to support runtime observation/monitoring adaptation, security decisions, trust building. They support new programming abstractions that deal with actions and interactions rather than state transformations.
- Events may have associated evidence, for example the sensor generating low level events, the algorithm used to extract information (colors, shapes, faces, sound patterns, ...), rules used to infer higher level events from lower level events (location of a person, end of lecture), or human input.

2.3 Challenge Areas

There are many challenges to realizing the full promise of event-based semantics. The overall challenge is to develop general mathematical models together with domain specific refinements that are both natural and expressive. Beyond models its is crucial to develop logics and reasoning principles. Finally these models and logics need tools to make them usable for analysis, synthesis, and transformations. Below we discuss some of things an appropriate event model should capture and some of the issues that will be faced in doing so.

Identifying, Modeling and Reasoning About Interdependencies. It is critical to enable increasing dynamic (computer) control that is safe and without unpleasant surprises. Consider for example, a power grid versus a transportation system. One the one hand, some elements of the transportation system depend on the power grid: trolly cars, fuel pumps, logistics planning. On the other hand changes in the transportation system may affect loads on the power grid. How can event partial orders combined with event timelines enable effective modeling and analysis? For example, an event based model could express dynamic consequences of dependencies, not just static relations.

Dealing with Time, Space, Scale, and Uncertainty. Time resolution of observation and actions. Events must be communicated in time to be useful. An example is sub second control on a power grid. Local event and control models may change over time. One example is different aircraft flight modes—takeoff, landing, or cruising. Another example is traffic control for high density landing of aircraft, weather and traffic patterns can change things substantially. Thus we must be able to model and reason about changes of event structures over time.

Systems operate at multiple time scales, for example realtime control of a single device in the context of scheduling of train/air traffic or coordination of human activity. An

event-based semantics is needed that supports reasoning at each scale and integrating multiple scales.

Event hierarchies and rules for deriving high level events from lower level evidence or refining high level events to lower level events such as actions are needed.

Privacy Issues for Human Centric CPS. Access to dynamic data raises new issues as patterns over time, together with context, can enable unexpected inference of information. An example: the water company may monitor patterns of water usage, for the purpose of optimizing flow control. Maybe be able to detect activity such as shower, toilet flush, running a dishwasher. If the electricity company and the water company shared information, it might be possible to infer more refined differences. Electricity patterns for a dishwasher might be different than those for a washing machine. Clearly there is the possibility of invasiveness if such inferences are made (and exposed) .

Notice that the dynamic data can be modeled as event streams and transformations on event streams can be used to control what information is exposed. Much work developing formal 'threat' models (knowledge context and ability of entity accessing data) is needed to realize this possibility.

Composition Composing is not just putting things together in parallel! It is also necessary to provide a means of interaction, and a means of constraining possible interactions. For example,

- What network and communication protocols are needed to enable interacting with physical systems?
- What coordination primitives are needed to describe event-based compositions that involve physical systems?
- What properties of the components and their composition are important?

Incomplete specifications are often more elegant and easier for a designer or implementor to work with, but they are generally not composable—as the missing information leaves open the possibility of interference or unexpected combined behaviors. Arguments for composition properties typically assume all events are known, while in a given event model some information will be implicit in the model. Assume/guarantee formalizations can help, but when composing using multiple models it will be crucial to make explicit all information relevant to the composition. A possible approach to cross model composition is to develop meta-models that make explicit model assumptions.

Another aspect of composition is composing evidence—proofs, statistical confidence levels, trust. For example, low level events or event streams may be combined an abstracted to infer higher level events. Event models are a good basis for thinking about situation awareness, and it may be important to know how and event was detected, before taking action.

New Models for Thinking About Things Top to Bottom. Currently embedded systems have their control loop in the hardware or a realtime operating system. This does not scale and does not work well in open systems. How can thinking in terms of events lead to better models?

New Languages Based on New Models of Computation and Interaction. Languages are needed for event-based requirements, executable specification, composing, monitoring, and even programming. Considerations include

- the ability to change the way instructions/descriptions are interpreted
- scoping visibility and effects of actions
- containing effects of errors or unexpected events—both physical and cyber
- programming concepts with resource sensitivity built in
- what can be monitored, detected and/or controlled?

3 Related Work

In the following we review a sampling of work related to event-based semantics.

Events Vs. State, Partial Order Vs. Interleaving. In [26,25] an argument is presented that the traditional computer science model of concurrent programming using state-based models and threads incurs unnecessary complexity and results in code that is difficult to debug. For sequential computation and function composition they work nicely. When deterministic sequential threads are composed in parallel they become non-deterministic and difficult to manage. A tag based signal model is proposed building on [24]. The domain of tags comes equipped with an ordering relation, events are tag-value pairs, and signals are sets of events describing incremental evolution of a system. Components modeled in terms of signals compose naturally. The model is elaborated to model both components and connectors, thus capturing interactions and also introducing the possibility of feedback loops. A mathematical theory based on topological concepts has been developed to give a compositional semantics to the components-connector wiring diagrams [27].

In [9] Clinger proves the existence of global times for event diagrams (a form of event partial orders) corresponding to possible interleavings. This provides an associated interleaving model allowing one to reason sequentially or about partial orders.

Rewriting logic [28,30] extends equational logic with local rewrite rules that model change over time. Proofs in rewriting logic can also be thought of as computations. Since rules are applied locally, a computation step may involve multiple parallel rewrites, while an equivalent computation carries these out one step at a time. In [29] it was shown that in a restricted class of rewrite theories modeling object / actor systems, there is an isomorphism between equivalence classes of proofs/computations and the event partial order generated by the computation.

In [36] an abstract interpretation of time is proposed to model systems involving preemptive scheduling. In this approach, models of the individual process are composed into a single time domain with the result being an infinite state timed automaton called a time domain automaton. Each state of such an automaton represents an equivalence class of all possible execution interleavings that result in that particular event ordering. Abstract interpretation combined with constraint solving techniques are used to make the model amenable to analysis.

Event Models for Actors. The actor model [20,2,1] is a model of concurrent and distributed computation based on asynchronous message passing. Actors are reactive entities that encapsulate state and control and interact with other actors only by sending and receiving messages. Events are the basis of semantics of actor languages and systems. Grief [18] introduced the notion of *event diagram* which captures the linear order of events at each actor and the causal order between message sends and corresponding receives. Baker and Hewitt [5] proposed a set of laws characterizing these event partial orders. In [33] a compositional notion of Actor Algebra is developed. Actor Algebra models include interfaces, specifications, event diagrams, and interaction paths with mappings between the different algebras.

PastTime Distributed Temporal Logic (PtDTL). PtDTL, a variant of PastTime Temporal logic was introduced in [32]. This logic reasons not over interleavings and linear sequences of past states, but over partially ordered sets of events causally in the past. As for event diagrams, events are located and the logic introduces epistemic operators that allow reasoning about what holds at the most recent causally previous state of another actor or process. The logic is used as the basis of an efficient algorithm for distributed monitoring.

Causal Logic of Events. Causal Logic of Events (CLE) [8,6,7] is a logic for distributed computing that has the explanatory and technical power of constructive logics of computation. CLE provides a proof technology that supports correct-by-construction programming based on the notion that concurrent processes can be extracted from proofs that specifications are achievable. A methodology for specifying distributed systems in CLE has been developed and implemented in NuPrl [3]. Requirements for a distributed system are expressed in terms of events, these requirements are then refined to collections of constraints called Message Automata (MAs) that imply the original requirements. MAs can be compiled to standard languages such as Java. Models of message automata are event diagrams, with events localized and the event order at each location a total order. Working bottom up, system properties can be inferred from MAs. Event classes and laws for composition allow specification and reasoning at a higher level of abstraction. The logical framework also supports timing properties, for example using variables that are trajectories of values rather than discrete values. The methodology has been applied to a variety of networking and security protocols.

Strand Spaces as an Event Model for Security and Location. Key exchange has logically simple goals, agnostic to communication concerns. In contrast, location protocols have quantitative goals, and models must consider transmission properties and use geometry. Strand spaces are a mathematical model that provides a special-purpose execution semantics, called Bundles, based on a causal partial order, that is complete for symbolic analysis of key exchange [16]. Reasoning about properties such as authentication or confidentiality makes combined use of causality and cryptographic properties. Strand spaces have been used to model and analyze a variety of security related protocols, including key exchange, contract negotiations, and secure payments systems.

Metric strand spaces are introduced in order to also reason about space and time. Bundles in a metric strand space have a distance and time elapse measures on some pairs of events that obey axioms reflecting by a model of transmission speed [19]. Secure

location protocols combine cryptography with the physics of message transmission. The cryptographic operations authenticate the principals and preserve confidentiality, while the physics of message transmission constrain their possible locations. In the case of metric strand spaces, the strand space model is enriched by associating a space-time location with each node. The strands follow the world lines of principals. Some bundles are compatible with the physics of message transmission – e.g. the maximum message transmission speed – while others are not. An assertion true in every bundle compatible with the physics is a valid conclusion of a secure location protocol.

CEL and strand spaces are similar in a number of ways. They share notions of causal order and the need to express limitations on the adversary. Strand spaces have a notion of unique origination that is similar to the CEL notion of nonce. The two formalism differ in their treatment of (logical) locality: in CEL locality encompasses multiple activities of a single principle or actor and may allow sharing of information across multiple threads/activities. In contrast the Strand space model explicitly isolates each activity of a principal (a strand) enforcing further localization.

Event Streams and Uncertainty. Event-based semantics is natural for many real-time embedded applications. In such applications the issue of temporal uncertainty is a common and challenging problem. Temporal uncertainty comes from the inherent restrictions in the underlying sensing layer, such as the temporal/spatial limitations of sampling, inaccurate clocks and unpredictable network latency. Although events occur instantaneously (dense time) in the physical world, an event occurrence often cannot be assigned a precise time owing to the above limitations. PTMON (Probabilistic Timing MONitor) [42,41] is a generic framework for incorporating various uncertainty models on event time stamps, developed in the context of monitoring timing constraints on event streams. A monitor task is formulated by timing constraints in a simple real-time temporal logic and satisfaction/violation of these formulas is checked at run time. Given a probability model of the temporal distance from event occurrence to event detection, timing constraints based on event occurrences can be transformed to those based on event detection. This transformation enables the early detection of timing constraint violations. Applications include real-time baggage tracking, wireless process control, remote monitoring, online multimedia downloading, and teleconference.

Grounding High-Level Event Definition. High-level event-based models require a precise notion of events in the model. A formal way of defining high-level events in terms of low-level system state helps to define a faithful abstraction for an event-based model. The logic of events and conditions (LEC) [22] is a two-sorted logic bridging the gap between state-based formalisms, commonly found in low-level models, and higher-level event formalisms. Conditions represent an abstract view of the system state, with primitive conditions being state predicates over the observable state variables in the system. Primitive events can also be directly observed during a system run. The use of LEC for event definition was developed in the context of run-time verification. The same separation of concerns used in run-time verification can be applied to high-level modeling in general. The basic approach is to provide an event-based model that uses high-level events as atomic building blocks, reducing the size and complexity of the model. The event definition layer provides grounding of the high-level model in the implicit low-level model. It can be used to establish a mapping between behaviors of the high-level

and low-level models, which can be used to demonstrate, without ever constructing the low-level model explicitly, that the high-level model is a faithful representation of the system.

Event Models for Pervasive Spaces. The Responsphere Infrastructure is a campus-level pervasive computing and communication infrastructure at University of California at Irvine (http://www.responsphere.org). It consists of a variety of sensors (video cameras, sensor mounted mobile robots, people counters, RFID, acoustic sensors, thermal and gas sensors) dispersed over approximately a third of the campus, connected via a variety of network and communication technologies (802.11, cellular, mesh, and power-line networks). It includes dense sensing in a few chosen buildings where it monitors all corridors, entries, exits, and public areas using cameras. In addition, some designated public spaces and laboratories are instrumented with RFID readers. Responsphere also includes mobile sensor mounted robots with communication capabilities that can be programmed for autonomous data collection. Responsphere serves as a test bed for developing a variety of pervasive functionalities, for example, using a mixture of video and RFID technologies to implement social policies of a shared common facility within a particular building. Examples include reminding people to switch off the coffee machine and conducting social experiments to study recycling behavior. In addition Responsphere has been used to conduct and monitor a variety of emergency drills such as building and region evacuations. The SATware System [21] (http://satware.ics.uci.edu) is a scalable middleware, that runs on Responsphere and provides seamless access to sensor and event level data. Applications access this information via a SQL style query language referred to as SATQL, at both the physical (e.g., raw sensor feeds) and semantic levels (i.e., at the level of entities, activities, and events). The key concept is that of a *virtual sensor* that empowers programmers to define and detect semantic concepts thereby realizing information abstraction. Virtual sensors are mapped (at run-time) to a graph of operators which are implemented over physical sensor streams. Challenges for management and programming of pervasive spaces include privacy and trustworthiness, evidence for judging semantic event reports, trading function for privacy, and self monitoring and adaptation.

4 Policy- and Goal-Based Operation of Autonomous Agents

There is a growing interest in autonomous agents that interact with and affect their environment, and have some ability to observe, reason, and adapt. As part of a larger system agents should also be able to compete for resources but also to cooperate for mutual benefit or to achieve an overall goal.

PAGODA (Policy And GOal based Distributed Autonomy) is a modular architecture for design of interactive autonomous systems. A PAGODA system is a collection of PAGODA nodes cooperating to achieve some mutual goal. A PAGODA node (agent) interacts with its environment by sensing and affecting, driven by goals to achieve and constrained by policies. The PAGODA architecture was inspired by studying architectures developed for autonomous space systems, especially the MDS architecture [15] and its precursors [31]. Software for deep space missions must be

- autonomous—operating remotely for extended time
- robust—operating under unpredictable conditions
- dependable—mission failure is costly

In PAGODA policy-based coordination is used at two levels: local modular combination of components making up an agents behavior; and coordination of a distributed system of agents constraining the possible interaction scenarios to meet end-to-end requirements.

The long term objective of the PAGODA project is to develop techniques for specification and analysis that take advantage of the modularity and the declarative nature of policy- and goal-based systems. PAGODA has been developed in the context of projects providing driving applications, including a rover (for example for exploration or patrol) [13,14] and software defined radios [40] supporting specific missions. Other potential applications include reactive/adaptive planners, cognitive radios, software assistants, and self-configuring systems.

Our approach is based on the Reflective Russian Dolls (RRD) model of distributed object reflection [29,34] which in turn is founded on the rewriting logic formal modeling framework [28,30]. In [34] a general approach to modeling policy-based coordination using RRD was presented. The question addressed by PAGODA is how to specify autonomous behavior that meets or achieves its goals (subject to constraints on external conditions) in a modular and declarative manner using models of its environment. Our solution is to factor the behavior into components, each with a specific role, that combine to achieve the desired result.

4.1 PAGODA Nodes

Figure 1 shows the principal components of a PAGODA node: a knowledge base (KB), a reasoner (R), a monitor (M), a learner (L), and a 'hardware' abstraction layer (HAL). These interact with each other and the environment under the control of a coordinator (C).

The knowledgebase (KB) is the centerpiece. It contains knowledge that is shared and updated by the remaining components. This knowledge includes a wide range of information:

- *Goals* that specify what the node or system is trying to achieve. A goal could be a very high-level goal such as carrying out a scientific experiment or tuning parameters to achieve a given quality of service; or lower level goals that correspond to actions that can be carried out.
- *Policies* that constrain the allowed actions / interactions of a node or system. A policy might reduce the number of choices for setting parameters, for example based on importance of different competing effects. Another policy might determine trade-offs between speed and power usage. Other policies might control aggregation and abstraction of information used locally or communicated to other agents.
- A *device model* that specifies the HAL interface: parameters/knob that can be set (effecting) and read (sensing) and their relationships. At the system level the model

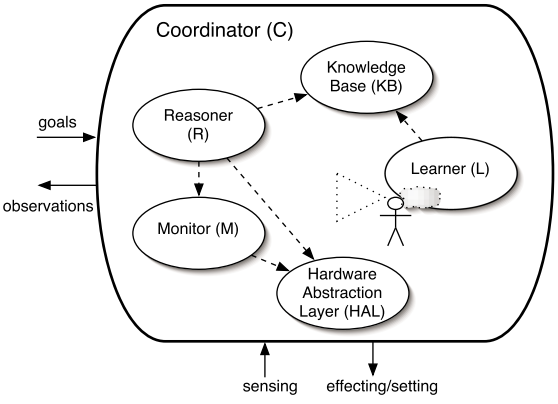


Fig. 1. PAGODA node architecture

should also specify how values sensed at different nodes can be combined to determine non-local system properties, and the relationships of such properties to higher level goals.

- An *environment model*, representing relevant features of the environment in which the node is operating, including information about other nodes. For a mobile node this could include terrain information or building maps.
- *Node state*, which includes values of variables determined by sensor readings and deduced from actions and information collected from other nodes. It also includes 'situation' information such as the stage in a complex task/mission or progress towards achieving a goal.
- *History*, a log of events–goals received, knob settings and sensor readings, monitor alerts, and so on.

The job of the *reasoner* component (R) is to determine proper parameter settings in response to goals requests: new goals, starting a new stage of a current goal, or alerts raised due to unexpected sensor values, indicating that adjustments need to be made. The reasoner uses information from the KB as a basis for its deductions: the device and environment model, the goals and policies, and the current state. When new parameter settings are determined, the reasoner also provides justifications such as what sensor values and/or what relationships from the device model were used to infer the new settings. This can be used for diagnostics if things don't go as expected. The reasoner also specifies sensors that should be monitored and conditions on sensor reading under which the reasoner to be alerted to take corrective action.

The *monitor* component (M) receives monitoring tasks from the reasoner, reads and evaluates specified sensors, and sends alerts to the reasoner when sensor readings are not within specified limits.

The job of the *learner* component (L) is to improve the model used by the reasoner to infer appropriate knob settings. In passive mode it observes events such as goals, settings, sensor readings and alerts and attempts to improve relationships specified by the model based on this information. A learner may also have an active mode where it is allowed to propose experimental settings and observe the results.

The *hardware abstraction layer component* (HAL) is an interface to the sensors and effectors used by the node. It plays the role of device driver, handling knob setting and sensor reading requests. In a real system the HAL might map requests to a format that is understood by the actual hardware, or even to a lower level abstraction layer. The intent is that these interactions should obey the 'physics' specified by the device model, but the node needs to be prepared for things to go wrong—some hardware component breaks, the environment is different than expected, it is being operated outside the expected operational mode, and so on.

The coordinator (C) controls message semantics for internal components and mediates interactions with the external world. The coordinator is responsible for ensuring specified relationships between the events (message deliveries) seen by different components, and for meeting logging and notification requirements. It also enforces component level synchronization constraints (only delivering messages for which the component is enabled). The coordinator actions are specified declaratively by policies. Note that coordinator policies are similar in spirit, to policies used by the reasoner, but different in detail.

Each PAGODA component type has an interface specified using events. The semantics of given component is an event-based semantics in the spirit of the Actor Algebra discussed in Section 2. This enables event-based composition of components and their semantics. Composition with a coordinator can be treated as a vertical composition in the spirit of [12].

This architecture provides a simple means of plugging in different component instances. PAGODA node components interact with other node components based on component type not on component instance identity. Thus it is easy to have multiple reasoners, knowledge bases, learners, etc., by simply modifying the coordinator policy to choose appropriate component instances. Different reasoners might be appropriate for different situations or goals, knowledge might be split into categories and stored in separate KB instances, or two KB instances might contain knowledge at different levels of abstraction appropriate for different situations.

Additional components types could be easily incorporated. For example a component capable of knowledge abstraction or aggregation could be invoked from time to time by the coordinator to infer higher-level information from sensor data or information received from peers. Such a component could be used to raise the level of abstraction at which the reasoner or learner operates.

5 Interactive Agents

Much has been written contrasting interactive computation and other models such as Turing machines and logic programming [37,38,39]. Our focus is on modeling and reasoning about the capabilities enabled by interactivity.

An interactive agent must be aware of its surroundings, and it may also affect its environment. It may need to negotiate, cooperate, or compete. A formal framework for modeling *interactive agents* was introduced in [35]. The framework was based on the need to consider the following features in the design of interactive agents.

- An agent has a boundary consisting of points of interaction with the environment. From the outside only what crosses the boundary is visible. Interaction points could be sensors, such as light detectors or thermometers, effectors such as switches or dials, or message queues for exchange of messages with other agents.
- An agent has actions that it can execute. It may also have goals, knowledge (about its environment and itself), policies constraining actions, or strategies for achieving goals.
- Internally an agent may have multiple concurrent activities; observing and processing sensory information; refining goals to subgoals, choosing actions, executing actions; evaluating and analyzing results: did actions have expected effect? updating knowledge by learning and inference;
- Interactivity means internal processes must be interruptible.

The framework is based on rewriting logic and a reflective model of coordination for managing an agents activities. New forms of interaction are introduced to model both message and channel/signal based interactions, and to pave the way for modeling continuous interactions. The compositional interaction semantics of [33,12] is extended to handle the new forms of interaction. The aims of the framework include:

- a higher level means of specifying and understanding agent behavior
- a place to classify agents with different 'skills'
- a formal design space to represent a variety of design decisions and to study tradeoffs resulting from decisions such as adaptability vs. predictability;

One advantage of the proposed framework is that specifications are executable, allowing prototyping of designs at many stages. In addition, such specifications are formally analyzable using the Maude rewriting logic system, and connections with other formal systems.

Briefly, interactive agents are formalized as actor like objects with rules for communication by messages, interaction through interface points, and policies for coordinating activities, also represented as (sub)agents. What the agent reads at an interaction point is controlled by the environment. What the environment can read is controlled by the agent (by a write action).

An *interaction path* is a (possibly infinite) sequence of interactions (events as viewed from an imaginary external observer). Each computation of an agent (allowed by the rewrite rules) gives rise to a set of interaction paths consisting of the (non-silent) interactions labeling the transitions (rewrite rule applications). The observable semantics of an interactive agent is thus the set of interaction paths of its possible computations. This definition derives from earlier work developing interaction semantics for actors [33,12], ideas from Timed Data Stream semantics for the Reo coordination model [4], and signal event semantics [23]. Interaction semantics is similar in spirit to the Interactive Stream Languages of [17]. The ideas are also related to work on *interfaces* of reactive and concurrent systems such as, [10,11].

Interaction semantics is compositional both vertically and horizontally. The semantics of the horizontal (parallel) composition of two systems is done by zipping compatible paths, one from the semantics of each system. Two paths are compatible if their subsequences of complimentary interactions, such as out/write in one and in/read in the other

match. In the composed path, these interactions become silent transitions and disappear, and the remaining interactions are merged. (See [33] for details in the case of horizontal, actor-actor composition, and [12] for vertical, actor-metaactor, composition.)

6 Conclusion

Cyber physical systems (CPSs) are an emerging phenomena. These systems are often not only software intensive, but also are tightly integrated with physical system, leading to many new challenges for design and development. We have proposed event-based semantics as a semantic foundation for Cyber physical Systems. We discussed a variety of notions of event, essential features and challenges for developing event-based semantics for CPSs. We also sketched two compositional models, one for autonomous agents and one for interactive agents. The latter providing forms of interaction such as needed in CPSs.

References

1. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
2. Agha, G.: Concurrent object-oriented programming. Communications of the ACM 33(9), 125–141 (1990)
3. Allen, S., Constable, R., Eaton, R., Kreitz, C., Lorigo, L.: The Nuprl open logical environment. In: McAllester, D. (ed.) CADE 2000. LNCS (LNAI), vol. 1831, pp. 170–176. Springer, Heidelberg (2000)
4. Arbab, F., Rutten, J.J.M.M.: A coinductive calculus of component connectors. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) WADT 2003. LNCS, vol. 2755, pp. 34–55. Springer, Heidelberg (2003)
5. Baker, H.G., Hewitt, C.: Laws for communicating parallel processes. In: IFIP Congress, pp. 987–992 (August 1977)
6. Bickford, M.: Specification and derivation of distributed programs using a logic of events (2007), Invited lecture for Worshop on Event-based Semantics 2007, http://blackforest.stanford.edu/eventsemantics
7. Bickford, M.: Abstract sequential programs and a logic of events (2008), Invited lecture for Worshop on Event-based Semantics (2008), http://blackforest.stanford.edu/eventsemantics
8. Bickford, M., Constable, R.L.: A causal logic of events in formalized computational type theory. Technical Report Technical Report 2005-2010, Cornell University (2005)
9. Clinger, W.D.: Foundations of actor semantics. AI-TR- 633, MIT Artificial Intelligence Laboratory (May 1981)
10. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Ninth Annual Symposium on Foundations of Software Engineering (FSE), pp. 109–120. ACM Press, New York (2001)
11. de Alfaro, L., Henzinger, T.A.: Interface theories for component-based design. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, p. 148. Springer, Heidelberg (2001)
12. Denker, G., Meseguer, J., Talcott, C.L.: Rewriting semantics of distributed meta objects and composable communication services. In: Third International Workshop on Rewriting Logic and Its Applications (WRLA 2000). Electronic Notes in Theoretical Computer Science, vol. 36. Elsevier, Amsterdam (2000)

13. Denker, G., Talcott, C.L.: Formal checklists for remote agent dependability. In: Fifth International Workshop on Rewriting Logic and Its Applications (WRLA 2004). Electronic Notes in Theoretical Computer Science. Elsevier, Amsterdam (2004)
14. Denker, G., Talcott, C.L.: A formal framework for goal net analysis. In: Workshop on Verification and Validation of Planning Systems. AAAI Press, Menlo Park (2005)
15. Dvorak, D., Rasmussen, R., Reeves, G., Sacks, A.: Software Architecture Themes In JPL's Mission Data System. In: IEEE Aerospace Conference, USA (2000)
16. Fábrega, F.J.T., Herzog, J.C., Guttman, J.D.: Strand spaces: Proving cryptographic protocols correct. Journal of Computer Security, 191–230 (1999)
17. Goldin, D., Smolka, S., Attie, P., Sonderegger, E.: Turing machines, transition systems, and interaction. Information and Computation Journal 194(2), 101–128 (2004)
18. Greif, I.: Semantics of communicating parallel processes. Technical Report 154, MIT, Project MAC (1975)
19. Guttman, J.: Strand spaces: From key exchange to secure location, Invited lecture for Worshop on Event-based Semantics 2008 (2008), http://blackforest.stanford.edu/eventsemantics
20. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: Proceedings of 1973 International Joint Conference on Artificial Intelligence, pp. 235–245 (August 1973)
21. Hore, B., Jafarpour, H., Jain, R., Ji, S., Massaguer, D., Mehrotra, S., Venkatasubramanian, N., Westermann, U.: Design and implementation of a middleware for sentient spaces. In: Proceedings of ISI 2007 (2007)
22. Kim, M., Kannan, S., Lee, I., Sokolsky, O., Viswanathan, M.: Java-MaC: A run-time assurance approach for Java programs. Formal Methods in Systems Design 24(2), 129–155 (2004)
23. Lee, E.A.: Concurrent semantics without the notions of state or state transitions. In: Formal Modeling and Analysis of Timed Systems. LNCS, pp. 18–31. Springer, Heidelberg (2006)
24. Lee, E.A., Sangiovanni-Vincentelli, A.: A framework for comparing models of computation. IEEE Transactions on Computer-Aided Design of Circuits and Systems 17(12), 1217–1229 (1998)
25. Lee, E.A.: Concurrent semantics without the notions of state or state transitions. In: Formal Modeling and Analysis of Timed Systems. LNCS, pp. 18–31 (2006)
26. Lee, E.A.: The problem with threads. IEEE Computer 39(5), 33–42 (2006)
27. Liu, X., Matsikoudis, E., Lee, E.A.: Modeling timed concurrent systems. In: CONCUR (2006)
28. Meseguer, J.: Conditional Rewriting Logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
29. Meseguer, J., Talcott, C.L.: Semantic models for distributed object reflection (invited paper). In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 1–36. Springer, Heidelberg (2002)
30. Meseguer, J.: A rewriting logic sampler. In: Van Hung, D., Wirsing, M. (eds.) ICTAC 2005. LNCS, vol. 3722, pp. 1–28. Springer, Heidelberg (2005)
31. Muscetolla, N., Pandurang, P., Pell, B., Williams, B.: Remote Agent: To Boldly Go Where No AI System Has Gone Before. Artificial Intelligence 103(1–2), 5–48 (1998)
32. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Efficient decentralized monitoring of safety in distributed systems. In: ACM TOSEM (submitted, 2006) (invited papers)
33. Talcott, C.L.: Composable semantic models for actor theories. Higher-Order and Symbolic Computation 11(3), 281–343 (1998)
34. Talcott, C.: Coordination models based on a formal model of distributed object reflection. In: 1st International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MTCoord 2005) (2005)

35. Talcott, C.: A formal framework for interactive agents. In: Arbab, F., Golden, D. (eds.) Foundations of Interactive Computation (FInCo 2007). Electronic Notes in Theoretical Computer Science, vol. 203, pp. 95–106. Elsevier, Amsterdam (2007)
36. Tidwell, T., Gill, C.: Abstract interpretation of time for preemptive scheduling of cyber-physical systems. Position paper for Worshop on Event-based Semantics 2007 (2007), http://blackforest.stanford.edu/eventsemantics
37. Wegner, P.: Why interaction is more powerful than algorithms. CACM (May 1997)
38. Wegner, P., Goldin, D.: Computation beyond turing machines. CACM (April 2003)
39. Wegner, P., Goldin, D.: The church-turing thesis: Breaking the myth. In: Cooper, S.B., Löwe, B., Torenvliet, L. (eds.) CiE 2005. LNCS, vol. 3526, pp. 152–168. Springer, Heidelberg (2005)
40. Wirsing, M., Denker, G., Talcott, C., Poggio, A., Briesemeister, L.: A rewriting logic framework for soft constraints. In: WRLA 2006 (submitted, 2006)
41. Woo, H., Mok, A.K., Chen, D.: Realizing the potential of monitoring uncertain event streams in real-time embedded applications. Position paper for Worshop on Event-based Semantics 2007 (2007), http://blackforest.stanford.edu/eventsemantics
42. Woo, H., Mok, A.K., Lee, C.-G.: A generic framework for monitoring timing constraints over uncertain events. In: 27th IEEE International Real-Time Systems Symposium (2006)

Design and Deployment of Large-Scale Software-Intensive Systems in Urban Districts

Research Challenges toward Future Affluent Ambient Society

Teruo Higashino[1,2]

[1] Graduate School of Information Science and Technology, Osaka University
[2] Japan Science Technology and Agency, CREST
Yamadaoka 1-5, Suita, Osaka 565-0871, Japan
higashino@ist.osaka-u.ac.jp

Abstract. With the advance of ubiquitous computing and ambient intelligence, several thousands of sensors and mobile devices can collaborate with each other in order to collect sensing information in wide areas and distribute location-aware information in real-time. In urban districts, several types of ubiquitous applications will be deployed and used in parallel in near future. It is known that reliability and performance of such ubiquitous applications are strongly affected by node mobility, fluctuation of node density, data transmission mechanisms (protocols), and so on. Therefore, in order to design and deploy such ubiquitous applications in urban districts as societal systems, we must anticipate the behavior pattern (mobility) of pedestrians and vehicles in those areas and develop resilient design methodology for high-reliable deployment and management of ubiquitous devices in underlying wireless communication environments. Intellectual management of a large amount of sensing information in mobile wireless Internet environments is also becoming important. Here, we focus on large-scale mobile wireless ubiquitous systems in urban districts as complex software-intensive systems, and discuss about research challenges for their design and deployment.

Keywords: Software-intensive systems, ubiquitous systems, MANET, urban planning, software design methodology.

1 Introduction

Due to the progress of the Internet and wireless communication technology, mobile wireless communication systems/applications are becoming popular. Those technology will be used for indoor services, localization, location-aware services, and so on. Mobile Ad-hoc Networks (MANETs) are expected to be useful and important as a way for achieving future affluent ambient society. In near future, several types of ubiquitous systems/applications will be deployed in urban districts where several thousands of sensors and ubiquitous devices will collaborate with each other in order to collect sensing information in wide areas and

M. Wirsing et al. (Eds.): Software-Intensive Systems, LNCS 5380, pp. 116–131, 2008.

distribute location-aware information in real-time. This will make our life more convenient and affluent.

In order for mega scale ubiquitous systems to become popular as societal systems, high reliability and robustness are required. However, designing and deploying such mega scale ubiquitous systems are very complicated tasks since it is hardly possible to construct realistic testbeds in real world for performance evaluation and reliability checking. Moreover, constituents of such ubiquitous systems are not stationary, and it is known that their mobility models greatly affect their performance and reliability [1,2,6,9,11]. Therefore, we must carefully consider mobility of humans, mobile sensors, robots and vehicles. Thus there are strong demands for software design methodology which allows us to design, analyze and validate such ubiquitous systems/applications in simple and effective ways. In near future, a large amount of sensing information will be used in urban life. By combining different types of sensing information, we might be able to generate more useful information for our life. Intellectual management of a large amount of sensing information in mobile wireless Internet environments is also becoming important.

There are several research challenges in designing future software-intensive systems [8,24]. Here, we focus on designing mega scale ubiquitous systems in urban districts and discuss about research challenges for their design and deployment. The expected design methodology should consider the following points : (1) mobility influence of sensor/ubiquitous devices, (2) efficient and high-reliable deployment of ubiquitous devices, and (3) well-regulated urban planning for deployment of ubiquitous systems. In the following sections, we discuss about those topics in details.

2 Designing Mega Scale Ubiquitous Systems

2.1 Sensing Information in Future Urban Life

In near future, a large amount of sensing information will be used in urban life. For example, as environmental monitoring, meteorological sensors might be deployed in urban districts, and they collect temperature, humidity, wind direction in those areas. By analyzing a large amount of meteorological data and regional data such as building information, we might be able to cope with heat island phenomena in urban districts. We can expect to take measures to meet those situations based on sensing information.

Sensing old constructions such as old buildings, bridges and roads becomes much more important in near future. It might become one of the most valuable sensor applications. Such information will be useful for maintaining old highways/roads and detecting deterioration of old constructions. Sensing vital signs of humans also becomes important. Those information can be used for emergency medical services and monitoring of chronic diseases. Monitoring objects using cameras can be used for several purposes in urban areas and factories. For future factory automation, new types of sensors, monitors and mobile robots might be used.

More accurate GPS and wireless localization devices will be developed. Location information of moving objects/pedestrians will be more popular for future ITS and location-aware services such as shop advertisement services considering pedestrians' favorite, real-time route navigation services, autonomous traffic control in crowded passages, and so on. Observation of children and old persons in urban areas will also become useful for keeping emotional trust in urban life. Sensors can localize each moving object and estimate its moving speed and direction. On future roads, those collected data will be informed to approaching vehicles for road safety.

In the above environments, a large amount of sensing information will be collected and held in mobile wireless Internet environments. In order for managing those information, we need intellectual mechanisms for extracting useful information from different types of sensing information. Ensemble engineering is needed for adequate search, quick response and accurate location-aware services. Uncertain numbers of objects/users also need to be treated in those environments.

2.2 Ubiquitous Services on MANETs

In future ambient society, several ubiquitous services in urban districts are considered. Some examples are listed below.

- *Communication services* : Now wireless LAN devices are getting cheaper and smaller so that they can be embedded into cellular phones, PDA and small sensor/ubiquitous devices. A plausible story is that communication between two end mobile nodes in the same local region is performed through communication channels of cellular phones and wireless LAN by seamlessly switching the two communication channels depending on connectivity and usable bandwidth of the wireless LAN. This will be effective to avoid waste of network resources. In such situations, node mobility and adopted protocols strongly affect the quality of communication. The deployment of base stations (or relay nodes) also affects it.
- *Information delivery services* : MANETs might be used to gather/distribute several types of information such as advertisements in city sections, safety information for disaster relief and traffic congestion information to neighboring vehicles. In those services, in order to reduce the number of dissemination messages, position-based information dissemination may be used to deliver messages. In position-based information dissemination, each mobile node decides when it should distribute the received messages to its neighbors, depending on the (history of) positions of the node like geocasting [14,20]. In this case, node mobility might affect message delivery ratios and required dissemination intervals. By anticipating node mobility in advance, we might be able to find a suitable dissemination policy for achieving better information acquisition ratios with a smaller number of messages.
- *Mobile server services* : Suppose that mobile nodes have enough processing power and storage space in order to execute server/database programs directly on the nodes. Then they can act as servers (mobile servers) that

provide information stored on the nodes to their neighbors. Here, we call such services as *mobile server services*. A plausible service example is that a mobile node executes a weblog server program on the node, and other nodes nearby the server node will access to the weblog through MANETs. In urban areas such as stations, large shopping malls and baseball/soccer stadiums, people might simultaneously request similar information such as routes to specific destinations, bargain information in the malls and highlight scenes of their watching sports. To realize this kind of services on MANETs, content caching on mobile servers becomes a very important technology. This is effective to avoid traffic concentration and to improve content discovery ratios. This encourages us to design mobility-aware caching strategies.

2.3 Mobility Modeling in Urban Districts

As we described in Section 1, constituents of the above ubiquitous systems are not stationary, and it is known that their mobility models greatly affect the performance and reliability [1,2,6,9,11]. Therefore, in order to evaluate the performance of those systems and check their reliability precisely, we need to consider more realistic mobility models. However, the reality of mobility models is always a compensation for the complexity. In particular, modeling real movement of mobile nodes with high fidelity for evaluation of town-widely deployed networks needs detailed observation or survey of those people moving from place to place along streets. Obviously such observation is not easily carried out due to cost, privacy and some other reasons.

In Fig. 1, we show flows of pedestrians in front of Osaka railway station. Fig. 1(a) denotes Random Way Point (RWP) mobility where pedestrians move randomly. In Fig. 1(b), pedestrians also move randomly but follow the form of sidewalks. Fig. 1(c) shows a realistic mobility model called *Urban Pedestrian Flow (UPF)* [10,11], which is very close to real pedestrians' flows (in [10], we have shown that its estimation error is at most 8%). In UPF mobility, the number of pedestrians at each sidewalk is decided based on simple observation of pedestrian flows at multiple intersections, and the moving direction of each pedestrian at every

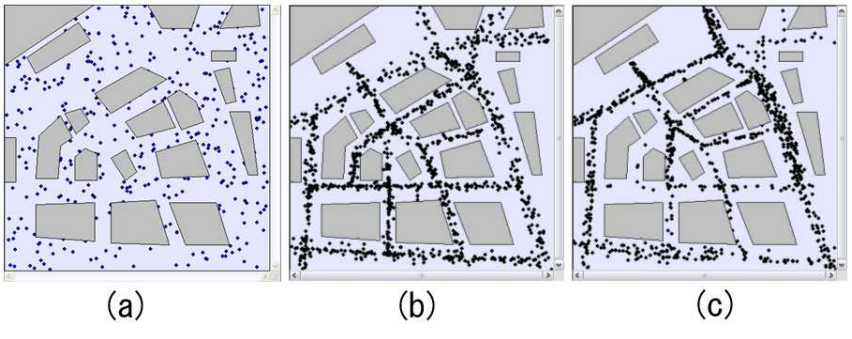


Fig. 1. Mobility Models : (a) Random Way Point (RWP), (b) City Section Mobility, (c) Urban Pedestrian Flow (UPF)

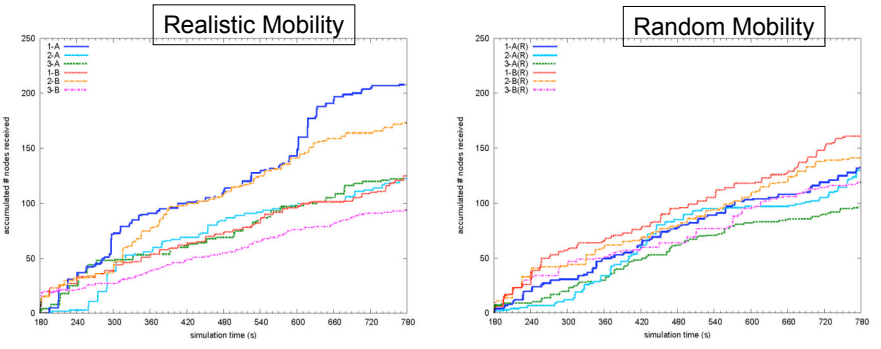


Fig. 2. Comparison of Packet Arrival Ratios between UPF and RWP

intersection is decided based on this calculation. Note that fluctuation of node density is rather large in Fig. 1(c) while it is uniform in Fig. 1(a) and Fig. 1(b).

We have designed a shop advertisement service for pedestrians around Osaka railway station where the bargain information of a department store is disseminated, and investigated the packet arrival ratios using several diffusion policies. We show the results in Fig. 2 where x-axis denotes the simulation time and y-axis denotes the number of nodes which have received the bargain information. We have changed the places of three base stations disseminating the bargain information by considering node densities obtained from UPF mobility in Fig. 1(c). We have also adopted two diffusion policies. As shown in Fig. 2, the arrangement of the base stations and diffusion policies do not give large influence for RWP (random) mobility since the node density is uniform. On the other hand, they give rather large influence for UPF (realistic) mobility since the node density varies depending on sidewalks and a suitable arrangement of the base stations achieves better performance. Detailed experiments about the difference of performance characteristics between RWP mobility and UPF mobility are shown in [11].

Recently many researches about transmission of traffic information among vehicles by using inter-vehicle communication without any load side infrastructures have been studied. In order to evaluate performance of such inter-vehicle communication, traffic simulators are widely used. However, in general, most of existing traffic simulators have been designed to reproduce statistically derived traffic flows (*macroscopic models*) since the goal of their usage is to evaluate traffic conditions such as prediction of traffic jams and traffic demands. Therefore, in such traffic simulators, distance between two following vehicles is uniformly implemented based on given statistical flow rates. If vehicles are running at the same interval, and if the interval is smaller than the wireless communication range, the success ratio of inter-vehicle communication between two sequential vehicles becomes very high. As the result, the success ratio of multi-hop communication among sequentially running vehicles also becomes high. In reality, vehicles are usually moving in groups of five or ten by following their front vehicle whose speed is relatively slow. Some vehicles among those vehicles might pass through from the current lane to the next lane. Thus, another groups of vehicles arise. In those situations, although

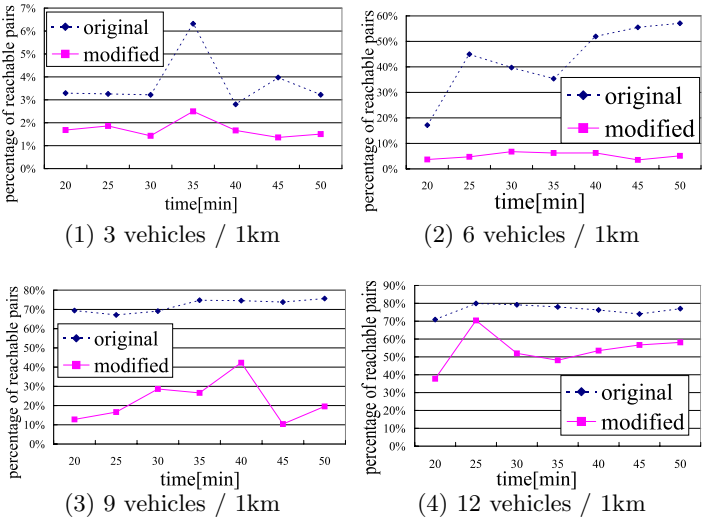


Fig. 3. Ratios of Reachable Pairs

the success ratio of multi-hop communication among sequentially running vehicles in a group is high, the success ratio of multi-hop communication between vehicles in different two groups becomes low if there are empty road sections between the two groups where no vehicles are running.

In [17], we have proposed a more realistic behavior model of vehicles (*microscopic model*) where we follow the basic vehicular mobility of an existing traffic simulator [16] so that the real bottleneck capacity and saturation flow rate on roads can be preserved. However, in the proposed model, for representing more realistic microscopic vehicular mobility, we modify the vehicular mobility where vehicles run at different speeds and distances between two vehicles vary. Each vehicle also selects its own favorite lane depending on its current speed and surrounding situation, and decides whether it should follow its preceding vehicles when it reaches the tail of a group of sequentially running vehicles. It can pass through when it reaches very slow vehicles. We have compared the average and distribution of vehicular distances in traffic flows reproduced by our microscopic model and its original macroscopic model with real ones, which can be obtained from observation for several cities' roads shown in Google Earth. The correlation coefficient between our microscopic model and real ones is 0.9411 while the corresponding correlation coefficient between the original macroscopic model and real ones is 0.3975. This shows that our microscopic model is sufficiently close to real ones.

In Fig. 3, we show the results of a simple experiment. Here, we say that a pair of vehicles is reachable when there exists a multi-hop wireless path between them. We have compared the macroscopic mobility (shown as "original" in Fig. 3) and proposed microscopic mobility (shown as "modified" in Fig. 3), and evaluated the success ratios of multi-hop wireless communication. In this simulaiton we set the maximum radio transmission range as $250m$. Then, we compared the ratio

of reachable pairs. We have used highway maps in Tokyo and Autobahn, and generated 3, 6, 9 and 12 vehicles per 1km in average through the simulation.

For the case of 3 vehicles per 1km (Fig. 3(1)), the ratio of reachable pairs is less than 7% for the both cases because the number of vehicles is too small to make multi-hop paths. On the other hand, for the case of 6 vehicles per 1km (Fig. 3(2)), the ratio of reachable pairs in the original mobility gets more than 50%, while that of the proposed microscopic mobility does not reach 10%. In the traffic flow made by the original simulator, there are a lot of vehicles running at almost the same distance (about $200-210m$). For the case of 9 vehicles per 1km, the ratios of reachable pairs in the both mobility are drastically different (Fig. 3(3)). For the case of 12 vehicles per 1km, the difference becomes rather small since the node density becomes high and the success ratios of multi-hop wireless communication in the microscopic mobility also become high (Fig. 3(4)). Thus, we can find that it is very important to consider more precise vehicular mobility in order to evaluate performance and reliability of inter-vehicle communication.

3 Research Challenges

3.1 Mobility Aware Design Methodology

In urban areas, several location-aware services are developing and/or planned. For example, the following typical ubiquitous services can be considered : (a) ITS services using inter-vehicle communication, (b) shop advertisement services for cellular phones with wireless LAN chips, and (c) emergency communication services in disaster districts. In such services, we cannot expect uniform node density nor random mobility. Most of pedestrians and vehicles follow traffic flows toward the same directions, and the node density varies depending on the locations since there are a lot of signals and obstacles in urban districts.

Although several research work considering mobility influence have been done in the areas of wireless communication networks, most of those work mainly focuses on the performance influence at lower network layers such as the MAC layer and the physical layer (for survey, see [1,2]). Those research work is very important to achieve high-reliable wireless communication in various environments. However, from the point of views of design and implementation of ubiquitous applications and large-scale software intensive-systems used in urban districts, we need to consider more upper layers' performance influence.

In our research group, we have developed mobile ad-hoc network simulator called *MobiREAL* [10,11,12,15] where it provides a new methodology to model and simulate realistic mobility of nodes and enables to evaluate MANET applications in more actual environments. It can simulate realistic mobility of humans and vehicles, and enables to change their behavior depending on a given application context. We adopt a probabilistic rule-based model called Condition Probability Event (CPE) model [11] to describe the behavior of mobile nodes, which is often used in cognitive modeling of human behavior. We describe mobility of those mobile nodes using C++ libraries designed for MobiREAL. The proposed model allows us to describe how mobile nodes change their destinations,

routes and speeds/directions based on their positions, surroundings (obstacles and neighboring nodes), information obtained from applications, and so on. It focuses on evaluation of MANET applications in urban districts, and applies it to evaluation of several ITS services and emergency communication services [18,19,21].

There are also similar research work which focuses on the upper layer design of ubiquitous applications. In order to develop high-reliable and autonomous ambient network services, more research work is needed so that ubiquitous applications used in urban districts have enough stability. Below, we will list typical research challenges in mobility aware design methodology.

- *Formal modeling techniques about movement of pedestrians and vehicles* : If we want to inform emergency information to people in disaster areas, we must carefully consider how people arrive at refuges via their evacuation routes, and make evacuation plans. If we want to design ITS systems for traffic safety, we need to precisely estimate how pedestrians and vehicles are moving at intersections and danger zones. There are some research work about behavioral characteristic of pedestrians and vehicles. Based on those research results, we need to formally specify the movement of pedestrians and vehicles. If we can use formal languages and models for specifying pedestriansf (vehicular) behavior in urban areas, the design and analysis of mobility-aware ubiquitous systems becomes much easier.
- *City-scale simulation environments* : Network simulators such as ns-2, GloMoSim, OPNET and QualNet have been widely used for evaluation of network traffic. Some of them can be used for evaluation of mega-scale mobile wireless networks such as campus-scale, city-scale and Internet-scale networks. However, as we mentioned above, most network simulators mainly focuses on the performance evaluation at lower network layers. Also, only simple mobility such as random based mobility is considered for performance evaluation of MANET applications. If we want to estimate arrival time and ratios of emergency information to people in disaster areas, we need simulation and/or emulation environments that can treat city maps and city-scale movement of pedestrians and vehicles.
- *More realistic emulation environments* : In some ubiquitous applications, we might want to reproduce the I/O of real applications in realistic environments because performance analysis based on statistical data does not always match user perception. For example, from statistical data, we cannot precisely imagine the difference of performance characteristic between video streams with stable 1 Mbps and those with fluctuating 0.8-1.2 Mbps. Therefore, if we can use APIs that allow real mobile terminals to connect to the simulated networks without modification of applications and protocols, and if we can reproduce the I/O of the target applications in real-time, the designer can imagine how the target ubiquitous applications work in real world. For this purpose, several hybrid approaches of simulation and emulation are studied, and some prototype tools such as TROWA, MobiNET and TWINE

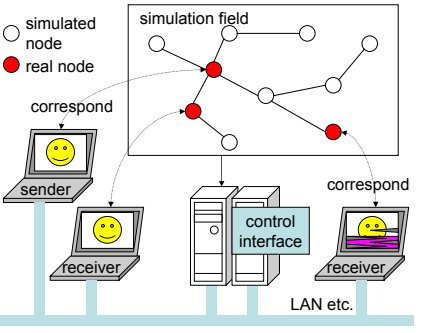


Fig. 4. Hybrid Architecture of Simulation and Emulation

have been developed [12,13,25]. Design of such emulation environments is one of important research challenges (see Fig. 4).

3.2 Deployment of Ubiquitous Devices in Urban Districts

Ubiquitous devices are attractive as means for providing intellectual services in urban districts. Some electric companies have been developing small ubiquitous devices, each of which contains CPU, memory, wireless communication device, rechargeable battery and so on (e.g. less than $1cm^3$ CPU device with ZigBee chip and button type battery). Although their computation power is rather small, those devices can be considered as small computers. In near future, several thousands of such small sensor devices are deployed in urban districts. Software for those sensors might need to be updated. Those devices have not only software for their own purpose but also the corresponding data and parameters. The values of such data and parameters may vary depending on location, node density and deployment, usable bandwidth, users' preferences, and so on. For example, two devices at different locations might have different map information in order to offer location-aware services. Thus, depending on locations and situations, different data and parameters might need to be set. This means that several thousands of small computers exchange programs/data through mobile wireless networks in the target areas. Manual setting and management of those programs/data values are complicated tasks and human errors might often occur. Therefore, we need more advanced middleware and tools for setting and management of those data/parameter values automatically.

In the research areas of grid computing and large-scale P2P applications, several types of middleware have been developed, which allocate programs and data to several thousands of remote computers and/or ask each of them to process small pieces of huge data. The grid middleware mainly supports to allocate programs and data to remote computers (e.g. [5]). The P2P middleware supports to distribute directory information to wide-spread P2P terminals and/or form suitable routes autonomously by avoiding bottleneck channels.

We can find several hints from middleware of grid computing and P2P applications in order to develop middleware for design and implementation of ubiquitous applications. Compared with grid middleware and P2P middleware, the middleware for ubiquitous applications in urban districts must have much more functions. It should have functions which manage mobile wireless networks. For grid computing and P2P applications, we can assume that two remote nodes can reach each other via TCP/IP connection. Although packet errors are considered, basically the target networks are considered as stable static networks. On the other hand, MANET applications connecting ubiquitous devices are sometimes disconnected and separated. They are not stable. They might be re-connected. Therefore, the middleware for ubiquitous applications should have functions for monitoring surrounding network environments and adjusting the corresponding parameter values for each device autonomously. More powerful functions for exceptional handling are also required.

Below, we will list typical research challenges in deployment of ubiquitous devices in urban districts.

- *Deployment of a large amount of ubiquitous devices* : As we discussed above, we need techniques for automatic deployment of ubiquitous devices and their parameter adjustment. If we have a plan to deploy million-order sensor devices in urban districts, we must consider how those sensors can communicate with each other by avoiding congestion. We need to develop intellectual techniques for cost-effective deployment of sensors. If we can estimate node density and mobility for the target areas, we might be able to find suitable deployment of sensors and routing of packets. Also, if we consider million-order sensor devices, there might be several faulty sensors and/or those with dead battery. Some sensors might not be able to store their assigned data because of network errors, memory errors and some other reasons. We also need techniques to find such faulty nodes quickly and recover them.
- *Seamless update of target software-intensive systems* : In general, most of software-intensive systems will be often updated after their deployment. Several thousands of people and vehicles might use ubiquitous devices even in a small city. Software update, parameter tuning and distribution of common data will be often carried out in urban districts. Since manual deployment and management need huge costs, it is needed that a few management staffs can control the target area's ubiquitous systems. Efficient mass update, parameter tuning and data distribution are one of key techniques for management of urban sized ubiquitous systems. For this purpose, we might be able to use mobile agent (and/or multi-agent) techniques which support million-order sensor devices simultaneously. Reprogramming techniques in wireless sensor networks (WSNs) might also be used (see [22] for survey of reprogramming in WSNs). We need techniques for updating the current version of software-intensive systems seamlessly. Since some software-intensive systems are used as societal systems, we might not be able to stop their operations. We must develop effective design methodology

for updating those devices without any stop of their operation. Building adaptive ubiquitous applications is one of the most interesting research topics.

– *Testbeds for emulating target systems* : When large-scale software-intensive systems are deployed in urban districts, several thousands of ubiquitous devices might be deployed on roads and walls of buildings. In order to make such systems societal ones, we need (1) testbeds for emulating the target software-intensive systems, (2) animating facilities for precise analysis of the behavior of people and vehicles, and (3) tools for checking whether target software-intensive systems can work well as societal systems under several environments of the target cities. Thoroughgoing preparations and rehearsals are definitely needed for constructing societal systems. As large-scale testbeds for WSNs, for example, there are MoteLab [23] and CitySense [3]. They usually provide management functions like online distribution of execution codes to mitigate maintenance costs. In [7], we have designed and developed an integrated environment called *D-sense* for supporting development of WSNs. D-sense supports protocol design by high-level design APIs. Also it provides seamless collaboration of its simulator and real networks for performance evaluation and distributed debugging. D-sense aims at comprehensive support of design, development and performance analysis. Further research for constructing more intellectual testbeds in mobile wireless network environments is needed.

3.3 Urban Planning for Deployment of Ubiquitous Systems

Currently, most of MANET applications are evaluated independently of other MANET applications. In wired networks, if two applications are deployed in parallel, they might share the common bandwidth and channels. This might make performance of the two applications decrease. However, by building additional channels, the bandwidth shortage problem might be solved. On the other hand, in mobile wireless networks, sharing the common channels might make heavy packet collisions. It might also be difficult to build additional wireless channels when the bandwidth shortage problem occurs since in general the number of usable wireless channels at each urban district is limited. Thus, in order to deploy multiple ubiquitous applications in urban districts, each ubiquitous application should consider a way to adjust the usage of given wireless channels. Thoughtless deployment of several ubiquitous systems at the same urban district and selfish bandwidth usage might confuse the operation of deployed ubiquitous applications.

We need some new software design paradigms for installing new applications into the current ones without stopping the current deployed applications. Ideally, when we deploy a new software-intensive system in a target city, it is desirable that we can automatically decide how the new software-intensive system should be installed from given city maps, their surrounding environments and underlying network resources, and that we can understand how much performance and reliability the software-intensive system can achieve when it is deployed. Also, if the new software-intensive system requires sensing data which have been already used

for other software-intensive systems, the collection/delivery methods of the sensing data might have to be re-arranged. Such re-arrangement should be also carried out automatically when a new urban planning is issued. We also need some tools for urban planning and management so that the urban planners can recognize the current situations. The well-regulated deployment planning as the common ICT basis in urban environments will become very important in near future.

Below, we will list typical research challenges in urban planning.

- *Middleware and tools for well-regulated urban planning* : When we make urban planning, enough scalability is needed so that mega scale devices can be easily treated on such environments. For this purpose, we need to develop middleware and tools for well-regulated urban planning for installing large-scale software-intensive systems in target urban districts. In urban districts, we can use both wired/wireless networks and cellular networks. As urban environments, we should consider roads, buildings, factories, shops, underground cites, malls, stations, bridges, parks, transportations and so on. The middleware and tools should support several millions of ubiquitous devices where different parameter values depending on target environments need to be suitably set to those devices. When developing such middleware and tools, we might be able to obtain some hints from grid middleware. It is true. However, logically grid computing assumes homogeneous computing resources. On the other hand, future software-intensive systems in urban districts take different actions depending on their locations and surrounding environments. Several types of heterogeneous sensing devices must be supported simultaneously. Thus, we need to develop new software technology for well-regulated urban planning.
- *Virtual worlds for understanding target software-intensive systems* : In general, if software-intensive systems are used in urban districts as societal systems, high reliability and availability are required. The urban planners and government officials might want to imagine how new software-intensive systems will be operated. For this purpose, usage of tools for constructing 2D- or 3D- virtual worlds might be useful. For example, we can use Second Life and SimCity as tools for constructing virtual towns. They provide some functions for constructing virtual towns. Several companies also sell precise 3D town maps for real big cities. However, in those tools and maps, we cannot handle wired/wireless communication infrastructures of target cities. We also cannot specify node mobility for people and vehicles in the cities. When we make evacuation planning in disaster areas, we need to specify usable communication resources and mobility of people and vehicles. If we can use tools for constructing virtual worlds based on specified communication resources and node mobility, they can make urban planners understand target software-intensive systems more precisely.
- *Techniques for preserving safeness and emotional trust* : If software-intensive systems are used as societal systems, malicious users might disturb their operation. Therefore, we definitely need mechanisms for detecting and removing faulty, selfish and/or malicious mobile devices autonomously.

When considering mega scale ubiquitous systems, we always need to assume that some ubiquitous devices might be faulty and/or malicious. Since a lot of ubiquitous devices are deployed, those devices should equip mechanisms for autonomously detecting faulty and/or malicious devices and removing them from the target system. Recently, Japanese government has a motto "ANSIN and ANZEN" for the development of the future ICT technologies. "ANSIN" and "ANZEN" mean "emotional trust" and "safety", respectively. It is obvious that those social systems require "safety". However, in order for residents in future ICT societies to live in affluent circumstances, they need to feel that the future services using such ubiquitous devices can really help the life of those residents. Thus, we also need to consider "emotional trust" when we make urban planning.

3.4 Further Research Challenges

Here, we give some other research challenges for developing future large-scale software-intensive systems in urban districts.

In urban districts, there are a lot of pedestrians and vehicles. When a pedestrian reaches a pedestrian crossing, its road surface sensors might detect such a pedestrian' approach and inform it to the surrounding vehicles. Simultaneously, such vehicles might receive information detecting invisible vehicles' approach from street corners. Such services are considered as popular future intellectual ITS services in many countries. Several types of information is transmitted at the intersections. Since those information need to reach target vehicles on time, the designers of those systems need to consider real-time constraints among several ubiquitous services so that safety information can reach the target vehicles and pedestrians on time. In such environments, vehicles and pedestrians issue several queries simultaneously. Some of queries are the same (or similar) and content caching on mobile nodes might offer high throughput. As we explained in Section 2.2, mobile server services might become popular when mobile nodes have enough computation power and storage space in near future.

Below, we will list typical research challenges concerning with the above topics.

- *Testing, verification and abstraction techniques* : We need to logically check whether deployed ubiquitous devices can work correctly as a total software-intensive system. It is important that we can check whether the total system can satisfy the constraints given in the specification. Since even if each device satisfies specified constraints, it is not clear whether the total system also do so. Currently, for example, model checking techniques are becoming popular, and they become useful for verifying the correctness of practical hardware and embedded systems. Some software model checking techniques are also proposed for guaranteeing the quality of complex software. There also exist proof techniques to guarantee given real-time constraints and performance requirements. Although those techniques are becoming very popular and powerful, for example, model checking techniques for FSM and TA might cause the so-called state explosion problem in general. Since we need

to consider mega scale ubiquitous systems, we must find new types of abstraction techniques so that we can prove the correctness of such mega scale distributed systems in reasonable time. Even if it is not easy to apply such proof techniques because of their complex structure, theoretical testing techniques might be useful. The assertion-based design methodology [4] might be also useful. We might need some hybrid techniques combining traditional verification (testing) techniques with some abstraction techniques (or some hierarchical analysis techniques). Thus, testing and verification techniques are becoming much more important in near future.

– *Intellectual data mining techniques in information-explosion era* : In urban districts, several sensing data are collected and those information is transfered to servers. It is very important to extract useful information from those enormous data sets and inform them to pedestrians and vehicles in urban districts. In general, people can only wait a few seconds when they issue their queries. If their answers cannot be replied in a few seconds, people will not use such ubiquitous services. Several services are simultaneously operated. So, each query need to be processed in a short period. Intellectual data mining techniques and caching mechanisms need to be proposed so that million-ordered transactions in a small area can be processed in real time. This is also a challenging research theme.

4 Conclusion

In this paper, we are discussing about research challenges for the design and development of large-scale software-intensive systems used in urban districts. The well-regulated urban planning can make affluent and comfortable living environments. Similarly, the well-regulated deployment and operation of ubiquitous systems can make high-reliable and affluent ubiquitous society. The research for design and deployment of large-scale software-intensive systems for affluent urban life is one of the exciting research themes for coming decades.

References

1. Bai, F., Sadagopan, N., Helmy, A.: The IMPORTANT Framework for Analyzing the Impact of Mobility on Performance of Routing Protocols for Ad hoc Networks. Ad. Hoc. Networks 1(4), 383–403 (2003)
2. Camp, T., Boleng, J., Davies, V.: A Survey of Mobility Models for Ad Hoc Network Research. Wireless Communications and Mobile Computing Journal 2(5), 483–502 (2002)
3. CitySense Project: CitySense - An Open, Urban-Scale Sensor Network Testbed, http://www.citysense.net/
4. Foster, H.D., Krolnik, A.C., Lacey, D.J.: Assertion-Based Design, 2nd edn. Kluwer Academic, Dordrecht (2004)
5. The Globus Toolkit, http://www.globus.org/

6. Higashino, T., Yamaguchi, H.: A Testing Architecture for Designing High-Reliable MANET Protocol. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 20–23. Springer, Heidelberg (2005) (Invited Paper/Keynote Speech)
7. Ikeda, K., Mori, S., Ota, Y., Umedu, T., Hiromori, A., Yamaguchi, H., Higashino, T.: D-Sense: An Integrated Environment for Algorithm Design and Protocol Implementation in Wireless Sensor Networks. In: MMNS 2008. LNCS, vol. 5274, pp. 20–32. Springer, Heidelberg (2008)
8. Information Technology for European Advancement (ITEA) Office Association: ITEA Technology Roadmap for Software-Intensive Systems, 2nd edn. (2004), http://www.itea-office.org
9. Le Boudec, J.-Y., Vojnovic, M.: Perfect Simulation and Stationarity of a Class of Mobility Models. In: Proc. of 24th IEEE Int. Conf. on Computer Communications (INFOCOM 2005), vol. 4, pp. 2743–2754 (2005)
10. Maeda, K., Sato, K., Konishi, K., Yamasaki, A., Uchiyama, A., Yamaguchi, H., Yasumoto, K., Higashino, T.: Getting Urban Pedestrian Flow from Simple Observation: Realistic Mobility Generation in Wireless Network Simulation. In: Proc. of 8th ACM/IEEE Int. Symp. on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM 2005), pp. 151–158 (2005)
11. Maeda, K., Uchiyama, A., Umedu, T., Yamaguchi, H., Yasumoto, K., Higashino, T.: Urban Pedestrian Mobility for Mobile Wireless Network Simulation. Ad. Hoc. Networks 7(1), 153–170 (2009)
12. Maeda, K., Nakata, K., Umedu, T., Yamaguchi, H., Yasumoto, K., Higashino, T.: Hybrid Testbed Enabling Run-time Operations for Wireless Applications. In: Proc. of 22nd ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS 2008), pp. 135–143 (2008)
13. Mahadevan, P., Rodriguez, A., Becker, D., Vahdat, A.: MobiNet: a scalable emulation infrastructure for ad hoc and wireless networks. ACM SIGMOBILE Mobile Computing and Communications Review 10(2), 26–37 (2006)
14. Mauve, M., Widmer, J., Hartenstein, H.: A survey on Position-Based Routing in Mobile Ad Hoc Networks. IEEE Network Magazine 15(6), 30–39 (2001)
15. MobiREAL Network Simulator Web Page, http://www.mobireal.net/
16. Mori, H., Kitaoka, H., Teramoto, E.: Traffic simulation for predicting traffic situations at Expo. 2005. R&D Review of Toyota CRDL 41(4), 45–51 (2006)
17. Nakanishi, K., Umedu, T., Higashino, T., Mori, H., Kitaoka, H.: Synthesizing Realistic Vehicular Mobility for More Precise Simulation of Inter-vehicle Communication. In: Proc. of 2nd IEEE Workshop on Automotive Networking and Applications (AutoNet 2007), CD-ROM (2007)
18. Nakamura, M., Urabe, H., Uchiyama, A., Umedu, T., Higashino, T.: Realistic Mobility Aware Information Gathering in Disaster Areas. In: Proc. of IEEE Wireless Communications and Networking Conf. 2008 (WCNC 2008), pp. 3267–3272 (2008)
19. Saito, M., Tsukamoto, J., Umedu, T., Higashino, T.: Design and Evaluation of Inter-Vehicle Dissemination Protocol for Propagation of Preceding Traffic Information. IEEE Transactions on Intelligent Transportation Systems 8(3), 379–390 (2007)
20. Stojmenovic, I.: Position Based Routing in Ad Hoc Wireless Networks. IEEE Communications Magazine 40(7), 128–134 (2002)
21. Umedu, T., Urabe, H., Tsukamoto, J., Sato, K., Higashino, T.: A MANET Protocol for Information Gathering from Disaster Victims. In: Proc. of 4th IEEE Int. Conf. on Pervasive Comp. and Comm. Workshops (PERCOMW 2006), pp. 442–446 (2006) (Invited Paper)

22. Wang, Q., Zhu, Y., Cheng, L.: Reprogramming Wireless Sensor Networks: Challenges and Approaches. IEEE Network 20(3), 48–55 (2006)
23. Werner-Allen, G., Swieskowski, P., Welsh, M.: MoteLab: aWireless Sensor Network Testbed. In: Proc. of 4th Int. Symp. on Information Processing in Sensor Networks (IPSN 2005), pp. 483–488 (2005)
24. Wirsing, M., Holzl, M.: Software-Intensive Systems, Report of the Beyond-the-Horizon, WG6 of IST-FET Coordinated Action (2007)
25. Zhou, J., Ji, Z., Bagrodia, R.: TWINE: A Hybrid Emulation Testbed for Wireless Networks and Applications. In: Proc. of 25th IEEE Int. Conf. on Computer Communications (INFOCOM 2006), CD-ROM (2006)

Formal Ensemble Engineering

J.W. Sanders[1] and Graeme Smith[2]

[1] International Institute for Software Technology
United Nations University, Macao, SAR China
[2] School of Information Technology and Electrical Engineering
The University of Queensland, Australia

Abstract. The 'ensembles' identified by the InterLink working group on Software Intensive Systems comprise vast numbers of components adapting and interacting in complex and even unforeseen ways. If the analysis of ensembles is difficult, their synthesis, or engineering, is downright intimidating. We show, following a recent three-level approach to agent-oriented software engineering, that it is possible to specialise that intimidating task to three levels of abstraction (the 'micro', 'macro' and 'meso' levels), each potentially manageable by interesting extensions of standard formal software engineering. The result provides challenges for formal software engineering but opportunities for ensemble engineering.

1 Introduction

Physical ensembles [6] incorporating potentially massive numbers of nodes, which interact with their physical environment and which may be adaptive and intelligent, offer a promising means of building many complex applications. A necessary condition for the widespread adoption and acceptance of physical ensembles, however, is trust in the dependability of such systems. Given the complexity of ensembles, it is our opinion that such trust can be achieved only using formal engineering methods.

The formal methods that exist today have not been developed to handle the complexity and scale proposed for physical ensembles. Hence, there is a need to develop new approaches. The nature of physical ensembles means these new approaches will vary depending on the level of observation applicable to the system being developed. Following Zambonelli and Omicini's summary of agent-oriented software engineering [21], we adopt three levels for observing ensembles: the *micro*, *macro* and *meso* levels (see Figure 1).

The *micro level* is applicable to ensembles which have a manageable number of components. For the purposes of ensemble engineering, we interpret that to mean 'distinct components': there may be a huge number of identical kinds (at this level of abstraction) of each component. Otherwise, engineering would be impractical. At this level, the behaviour of each component and each component interaction can be formally modelled and analysed. An example of such an ensemble is the system of sensors and actuators controlling a smart home designed, for example, for energy efficiency, or *assisted living*, *i.e.*, allowing an elderly or disabled person to live alone.

M. Wirsing et al. (Eds.): Software-Intensive Systems, LNCS 5380, pp. 132–138, 2008.

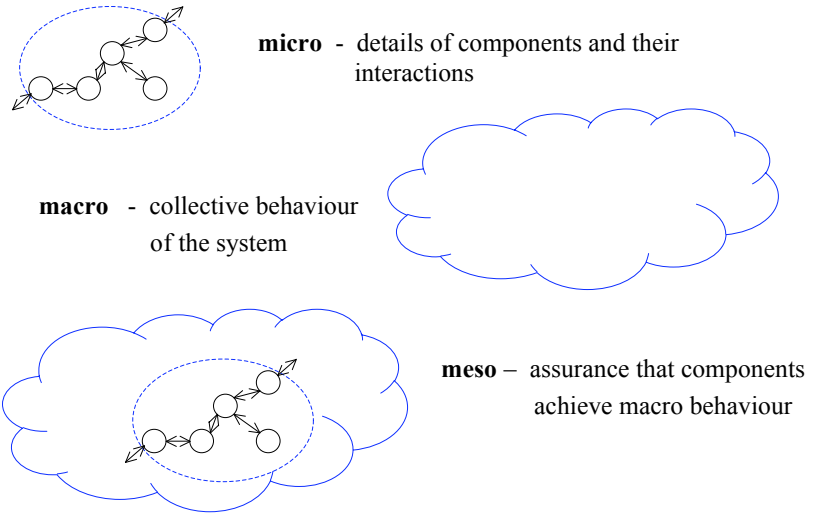


Fig. 1. Levels of observation

The *macro level* is applicable to ensembles comprising massive numbers of components, possibly distributed over a network and operating in a dynamic and uncontrollable environment. It is not feasible to engineer such systems in terms of individual components. Instead, means of engineering the collective behaviour of the components are required. An example of such an ensemble is an *ad hoc* sensor network deployed in an urban environment, *e.g.*, to monitor traffic jams and accidents and wirelessly communicate such information to drivers in the vicinity [16]. The collective behaviour in this case would be congestion-free traffic flow.

The *meso level* is seen as pertaining when an existing micro-level component is added to an existing macro-level system. In [21], concern centres on verifying that the deployment of a micro-level system within a macro-level one does not compromise the behaviour of either system. In our approach, the macro-level system *is* the specification of the whole system, the micro level contains the implementation, and the meso level embodies designs by which the members of the micro level achieve the behaviour specified at the macro level. For example in the sensor network, the meso level would contain structures to enable (GPS aware) vehicles to communicate with traffic sensors and each other. In top-down development of a system, the meso level bridges the gap between the macro and micro levels, showing what a micro-level component must achieve in addition to its unilateral micro-level behaviour. By requiring that the micro- and macro-level behaviours are not compromised, that returns us to the outlook of [21].

2 Micro-level Ensemble Engineering

The key challenge at the micro level of observation is the extension and modification of traditional formal methods. Much work has been done on formal

methods in the areas of continuous real-time [4,22], probability [10,8] and mobility [11,3], all areas of importance for physical ensembles. However, other areas of equal importance have received only limited attention; in particular, the areas of spatial location, autonomy and intelligence, and adaptation.

2.1 Spatial Location

The spatial location of components in some ensembles is vital. For example, in an industrial manufacturing setting the precise location and orientation of a robot working in a team with other robots is a necessary part of its specification if collisions are to be avoided and cooperation achieved. Other applications such as claytronics [5] or free-flight air traffic control [2,15] also require precise locations of components to be specified.

For some applications a discrete notion of space may be sufficient, in others continuous space may be required. In either case, new formal methods should be developed where space is a first-class concept, rather than just modelled. Approaches to incorporating real-time into formal methods should offer some guidance.

2.2 Autonomy and Intelligence

In most existing formal methods, components are reactive. They do not have goals and plans that enable them to act autonomously. Although much work has been done in the artificial intelligence (AI) community on goal-oriented decision making, there has been little integration of this work with formal methods, or with software engineering in general. Most often, AI techniques, when used, are introduced during the implementation phase of a project, rather than during high-level requirements analysis and design phases.

Incorporating AI techniques with formal methods is essential if we are to promote their consideration at the highest levels of system abstraction. Possible approaches include new formal methods based on agent-based approaches [20] and machine learning [13], or on non-standard logics, such as fuzzy logic [7] or non-monotonic logics [1], which can be used to model intelligent decision making processes.

2.3 Adaptation

Components in ensembles will need to adapt their behaviour to respond to unforeseen changes in their environment. It is possible to model changes in behaviour within a single specification using, for example, appropriate operators for combining behaviours [19]. It is also possible to do so if the state spaces of the various adaptations have a uniform abstraction [17]. However, deciding on the kinds of changes which are allowable for truly adaptive components is difficult.

Research on changing high-level requirements of real-time specifications has shown that all such changes can be modelled as a sequence of refinements and a

minimal set of basic rules [18]. Similarly, changes to component configurations in an object-oriented setting can be modelled as a sequence of refinements and a minimal set of rules [9]. Hence, it seems feasible that a formal calculus of specification change could be developed.

By establishing a formal relationship between pairs of specifications, such a calculus would enable us to reason about changes to a given specification. In particular, it would enable us to determine the effect a change has on established properties of the specification. This would potentially enable us to reason about the effects of adaptation, and to determine the limits of adaptability that would maintain critical properties at both the component and system levels.

3 Macro-level Ensemble Engineering

A macro-level system can be *specified* simply as a combination of all the micro-level components (described unilaterally), conjoined with a condition ensuring that the result behaves as desired. Typically that condition captures behaviour that is thought of as being *emergent*: not a consequence of the behaviours of the unilateral micro-level components. Without it, the specification would allow undesirable behaviours resulting from the undisciplined interaction of components at the micro level. Such a specification trades clarity for any hint of implementation strategy. It is at the meso level that the emergent condition is to be achieved, somehow, from the micro components.

In the traffic-sensor example, the micro-level might describe the system in terms of components (cars, public transport and commercial vehicles) 'interacting' on the roads; there are many instances of each component. Then the macro specification would contain those, mediated by a predicate ensuring the smooth flow of traffic. Such behaviour is of course emergent when viewed from the level of an individual vehicle.

It is to be expected that, because of the huge number of components in an ensemble, macro-level behaviour is captured using distributions (in the sense of statistics) and even notions of convergence in space or time (to describe the effects of adaptability in achieving what might be termed 'societal stability'). That must in turn affect the definition of conformance of a design to its specification. But conformance 'at a certain confidence level' may not sit well with abstraction [14] and so must be investigated.

4 Meso-level Ensemble Engineering

The goal of the meso level is to ensure that refinement holds when a macro-scale specification, *MacroSpecification*, is augmented with a micro-scale component, *MicroComponent*, as part of the design process. In terms of the symbol $\sqsubseteq$ for 'valid refinement',

$$MacroSpecification \sqsubseteq MicroComponent \wedge MacroSpecification' .$$

Now the right-hand side specifies a design in which the ingredients combine to achieve the ensemble specification on the left. In the traffic-sensor example, a design achieving free traffic flow might incorporate the relay of information from traffic sensors to vehicles which use GPS and data from neighbouring cars to regulate their velocity (speed in current route or change of route) to avoid traffic jams. It is important to acknowledge the role played by human drivers in the adaptability necessary to achieve emergence: each smart car might offer its driver a choice of possibilities and different driver preferences might be expected to substitute for the randomisation required in network routing algorithms to avoid repeated blockages.

Designs at the meso level are complicated by the fact that components in both the micro and macro levels may be mobile and hence the systems may merge, blurring their boundaries. From the viewpoint of just the micro level, it would be usual to place assumptions on the environment of a component. Similarly, to engineer a macro-level system requires assumptions about the interactions between the components at the meso level.

To exploit the proposed formal approach, we need to formalise the allowable meso-level interaction patterns and verify that the behaviour and assumptions of the components in the micro-level system conform to these patterns. Suitable formalisms could be built on process algebras, especially those supporting mobility [11,3], adding elements of, for example, game theory to capture the more complex behaviour possible with autonomous, intelligent components. Also relevant to such formalisms is current work on languages for orchestrating distributed systems [12].

5 Summary

To engineer physical ensembles formally, we propose extensions to traditional formal methods and the way they are applied at three levels of observation:

1. At the *micro level* where we have a manageable number of distinct components, we require extensions to existing formal methods which
   - have a first-class concept of *spatial location*,
   - incorporate AI techniques to capture *autonomy and intelligence*, and
   - have theories, beyond refinement, relating specifications in order to reason about *adaptability*.

2. At the *macro level* where the number of components is massive and it is not feasible to think in terms of individual components, we need a notion of an *emergence condition* which captures the desired collective behaviour of the system and, importantly, rules out undesirable behaviours. We also need to investigate the conformance of designs to specifications where behaviour is statistically defined.

3. At the *meso level* where we introduce micro-level components as part of designing a macro-scale system, we need formalisms which allow us to capture

and reason about complex interaction patterns. It is at this level that we move from the clarity of emergence predicates at the macro level, to the strategies employed to satisfy those predicates at the micro-level.

Acknowledgements. The authors thank Kirsten Winter for discussions on these ideas and comments on an early draft of this paper.

References

1. Brewka, G., Dix, J., Konolige, K.: Nonmonotonic Reasoning - An Overview. CSLI publications (1997)
2. Byrne, G.: Is it time to give airlines the freedom of the skies? New Scientist 2351, 12 (2002)
3. Cardelli, L., Gordon, A.: Mobile ambients. Theoretical Computer Science 240(1), 177–213 (2000)
4. Fidge, C.J., Hayes, I.J., Martin, A.P., Wabenhorst, A.K.: A set-theoretic model for real-time specification and reasoning. In: Jeuring, J. (ed.) MPC 1998. LNCS, vol. 1422, pp. 188–206. Springer, Heidelberg (1998)
5. Goldstein, S.C., Campbell, J.D., Mowry, T.C.: Programmable matter. IEEE Computer 38(6), 99–101 (2005)
6. Hölzl, M., Wirsing, M.: State of the art for the engineering of software-intensive systems. InterLink Deliverable Number D3.1 (2007), http://interlink.ics.forth.gr/central.aspx?sId=84I238I744I323I344283
7. Klir, G.J., Yuan, B.: Fuzzy sets and fuzzy logic: theory and applications. Prentice-Hall, Englewood Cliffs (1995)
8. Kwiatkowska, M., Norman, G., Parker, D., Sproston, J.: Verification of real-time probabilistic systems. In: Merz, S., Navet, N. (eds.) Modeling and Verification of Real-Time Systems: Formalisms and Software Tools, ch. 8, pp. 249–288. John Wiley & Sons, Chichester (2008)
9. McComb, T., Smith, G.: A minimal set of refactoring rules for Object-Z. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 170–184. Springer, Heidelberg (2008)
10. McIver, A.K., Morgan, C.C.: Abstraction, Refinement and Proof for Probabilistic Systems. Monographs in Computer Science. Springer, Heidelberg (2005)
11. Milner, R.: Communicating and Mobile Systems: The π-Calculus. Cambridge University Press, Cambridge (1999)
12. Misra, J., Cook, W.R.: Computation orchestration: A basis for wide-area computing. Software and Systems Modelling 6(1), 83–110 (2006)
13. Mitchell, T.M.: Machine Learning. McGraw-Hill, New York (1997)
14. Morgan, C.C., McIver, A.K., Sanders, J.W.: Probably Hoare? Hoare probably! In: Davies, J., Roscoe, A.W., Woodcock, J.C.P. (eds.) Millenial Perspectives in Computer Science, pp. 271–282. Palgrave (2000)
15. RTCA Task Force 3. Final Report on Free Flight Implementation. RTCA Inc. (1995)
16. Saito, M., Tsukamoto, J., Umedu, T., Higashino, T.: Design and evaluation of inter-vehicle dissemination protocol for propagation of preceding traffic information. IEEE Transactions on Intelligent Transportation Systems 8(3), 379–390 (2007)

17. Sanders, J.W., Turilli, M.: Dynamics of control. In: Theoretical Aspects of Software Engineering (TASE 2007), pp. 440–449. IEEE Computer Society, Los Alamitos (2007); Expanded version available as: UNU-IIST report 353, http://www.iist.unu.edu
18. Smith, G.: Stepwise development from ideal specifications. In: Edwards, J. (ed.) Australasian Computer Science Conference (ACSC 2000). Australian Computer Science Communications, vol. 22, pp. 227–233. IEEE Computer Society Press, Los Alamitos (2000)
19. Smith, G.: Specifying mode requirements of embedded systems. In: Oudshoorn, M. (ed.) Australasian Computer Science Conference (ASCS 2002). Australian Computer Science Communications, vol. 24, pp. 251–258. Australian Computer Society (2002)
20. Wooldridge, M.: An Introduction to MultiAgent Systems. John Wiley & Sons, Chichester (2002)
21. Zambonelli, F., Omicini, A.: Challenges and research directions in agent-oriented software engineering. Autonomous Agents and Multi-Agent Systems 9(3), 253–283 (2004)
22. Chaochen, Z., Hoare, C.A.R., Ravn, A.P.: A calculus of durations. Information Processing Letters 40, 269–271 (1991)

Structured Interacting Computations
(A Position Paper)*

William Cook and Jayadev Misra

The University of Texas at Austin

Abstract. Today, concurrency is ubiquitous, in desktop applications, client-server systems, workflow systems, transaction processing and web services. Design of concurrent systems, particularly in the presence of communication failures, time-outs and interrupts, is still difficult and error-prone. Theoretical models of concurrency focus on expressive power and simplicity, but do not provide high-level constructs suitable for programming. We have been developing a theory, called *Orc* (for orchestration), and its practical applications. In this paper, we describe our philosophy in designing Orc. The guiding principle is to structure a concurrent program in a hierarchical manner, and permit interactions among subsystems in a controlled fashion. The interactions are described by *value passing*; the mode of communication (i.e., whether the value is passed over a channel or kept as shared data, etc.) is left unspecified.

1 Introduction

1.1 Nature of Concurrency

Today, concurrency is ubiquitous, in desktop applications, client-server systems, workflow systems, transaction processing and web services. Concurrency will continue to permeate many areas, such as manufacturing and inventory control, medical applications, command and control systems, and embedded systems in automobiles and avionics. These trends will require all but the simplest applications to be structured as part of an interacting computation. In particular, such computations must coordinate multiple activities, manage communication and handle failures, time-outs and interrupts as they leverage concurrent services.

The internet has brought concurrency to the fore, promising a truly global computer where all services and data are available to all users at all time. In particular, an application ought to be able to call upon remote services, utilizing remote data, and have the results be piped to yet other remote services. The notions of data and service migration, application discovery, and downloading of services for local executions should be totally transparent to the users and other applications in a global computer. The hope is that, ultimately, the entire mankind is connected, not just in sharing common news, but in collaborating on various activities which span time scales from milliseconds to decades.

* Supported by NSF grant CCF-0811536.

M. Wirsing et al. (Eds.): Software-Intensive Systems, LNCS 5380, pp. 139–145, 2008.

This promise of the internet has not been met, nor even (discounting the pun) remotely met. Today, the internet functions merely as a communication service provider. It is typically used for downloading large amounts of data (news pages or video, for instance), or for point-to-point communication, as in email or remote application invocation (whether implemented synchronously or via store-and-forward). True internet computing will invoke multiple available services concurrently, initiate data and program discovery and exchange, allow time-outs and degrade gracefully under failure.

Concurrency in embedded systems, as in video games, automobiles and avionics, are yet more specialized. The applications (and the physical components) are expected to meet stringent real-time constraints and handle faults without end-user intervention. Synchronization among components is a common activity, though it is far less so among components of a wide-area system like the internet.

A general model of concurrency will have to encompass concurrent computations that may interact at a frequency of milliseconds (as in avionics) to those in which communication may occur once in a month (as in workflow systems).

1.2 Interaction Mechanisms

Historically, the form of interaction among components has been influenced by the available hardware. The initial applications of concurrency were in the design of specialized systems such as operating systems. Simple locking mechanisms, such as semaphores, were adequate for such applications, to prevent simultaneous access to shared data on a single computer. These methods worked well for communications among a limited number of components. Later, more sophisticated techniques, such as critical region, monitors and transactions have been employed to restrict access to shared data.

There have been major theoretical advances in recent years in describing interactions. Several process algebras, such as CSP [6], CCS [8], π-calculus [9] and join calculus [1], have been developed with the sole purpose of describing interactions. These algebras have not only inspired practical developments, such as polyphonic abstractions in C# [1], but also established the capabilities of interacting systems. In particular, π-calculus can simulate λ-calculus in a natural manner; thus, interactions alone in that system can simulate the power of a Turing machine.

Transactions [5] have been particularly effective in terms of practical developments, because they place far less burden on the programmer in managing concurrency. The programmer writes sequential code, and then declares a code fragment to be a transaction. She imagines that during execution of a transaction, it is the only piece of software executing in the world. No other software can have any influence on it during this time. This illusion, known as *atomicity*, is implemented by a transaction processing system which restricts (or delays) certain requests to shared data. Further, elaborate precautions are taken to ensure that shared data is in a legal state after canceling of a transaction, because the transaction may be canceled after changing the value of data. A transaction is required to explicitly commit in order to effect permanent state changes in

data. These implementation details are hidden from the programmer, making transactions appear as non-interacting, atomic code fragments.

There are still several outstanding issues in the treatment of interactions. While the theoretical models, such as π-calculus, seem to capture the essence of interactions, it has proved difficult to translate these ideas into a practical setting; join calculus is an effort in this direction. For many applications, the programmer should not have to work at the level of channel-based communication or primitive synchronization.

Transactions pose a different set of problems. The semantics of transactions, particularly nested transactions, have not yet been adequately defined. Integration of transactions with other concurrency features, such as locks and communicating processes, have not been addressed. Integration with real-time features (and, hence, deploying transactions for embedded system design) remains a problem.

2 Structured Concurrent Programming

We have been working on a theory, called *Orc* (for orchestration), and its practical applications[1]. We describe below our principle in designing Orc. The guiding philosophy in the design is to permit structuring the program in a hierarchical manner, while permitting interactions among subsystems in a controlled fashion. The interactions are described by *value passing*; the mode of communication (i.e., whether the value is passed over a channel or kept as shared data, etc.) is left unspecified.

2.1 Site

A cornerstone of Orc design is to permit *integration* of components. There are numerous software components designed by third parties that can be used in building an application. These components may be part of a general purpose library, or designed with the express purpose of solving a specific problem. A component could be a procedure to invert a matrix, compress a jpeg file, return the latest quote on a stock, or do a search of the internet (in the last case, the component is a giant piece of software, like the Google search engine). Typically, a component is called a *service*; we adopt the more neutral term *site*. Note that a site does not necessarily denote a web site, though a site could have a web interface. Orc has been designed to orchestrate the execution of sites.

An orchestration may involve humans, modeled as sites. A program which coordinates the rescue efforts after an earthquake will have to accept inputs from the medical staff, firemen and the police, and direct them by sending commands and information to their hand-held devices. Humans communicate with the orchestration by sending digital inputs (key presses) and receiving output suitable for human consumption (print, display or audio).

As described above, our sites are quite general. They need not be just functions in a mathematical sense. A site may return different values at different times.

[1] The Orc home page, at `http://orc.csres.utexas.edu/`, contains pointers to research papers and a prototype implementation.

A site could also possibly change the state of some system (imagine buying an airline ticket online; the site that implements this procedure changes the airline's database). A site need not be sequential code (consider an internet search engine). By permitting a very general definition of site, we expect to utilize all the software that are publicly available, as well as those that are designed for specific applications. In fact, the basic arithmetic and logical functions are also regarded as sites (this aspect of Orc is reminiscent of pure object oriented programming, like Smalltalk). So, we can remove considerations of data types from our theory, relegating them to sites. A practical system may implement primitive data types, or even more complex ones like XML [4], by inline code, thus mitigating any loss of efficiency. And, we can build our theory independent of the kind of data that are being manipulated or passed to and from sites.

We impose few restrictions on the interface of a site. A site is called like a procedure, and it is expected to return at most one value. There is no guarantee that a site actually returns a value. A site specification may include the amount of time taken for a site to respond, or there may be no such guarantee. The restriction that a site returns no more than one value, rather than a stream of values, simplifies the theory considerably. Further, a stream of outputs can be handled by asking the site to send one value in each call, and send acknowledgement for each value received, which prompts the next call.

Removing data types from Orc has the consequence that Orc programs are *stateless*. There is no point in storing any value, because Orc provides no machinery for manipulating data except through site calls. Therefore, the value returned by a site is never stored, but used as parameters of site calls. Although the Orc language itself does not support mutable state, sites are frequently stateful.

What happens to the value returned by a site? Suppose we call $CNN()$, site $CNN()$ responds with a news page, and there are no other site calls to be made. Effect of evaluating the expression $CNN()$ is to call the site and *publish* the value returned by it. Publication has no physical meaning; think of it as the result of computation.

2.2 Combinators

The next major design issue in Orc is to invent its *combinators*. A combinator in Orc combines two expressions to form another expression. We have primitive expressions, like $CNN()$, which are merely site calls. By applying the combinators, we can create arbitrarily complex expressions. Additionally, we create a hierarchical structure of the program which is amenable to inductive analysis.

For expressions f and g, we have three combinators: (1) symmetric composition, written as $f \mid g$, which allows independent execution of f and g; the sites called by f and g individually are called by $f \mid g$ and the values published by f and g are published by $f \mid g$, (2) sequential composition (also called *piping* or *push*), written as $f >x> g$, which initiates a new instance of g for every value published by f; the value is bound to name x in that instance of g, and the values published by all instances of g are the ones published by $f >x> g$, (3) asymmetric composition (also called *pull*), written as $f <x< g$, which evaluates

f and g independently, but the site calls in f that depend on x are suspended until x is bound to a value; the first value from g is bound to x, evaluation of g is then terminated and suspended calls in f are resumed; the values published by f are the ones published by $f <x< g$. A more complete description of the Orc language features may be found at [10,7].

We have a definition mechanism for expressions. A definition is of the form $E(p) \triangleq f$, as is standard in the style of declarative programming. The parameters of the definition, p, may appear in expression f. Additionally, name E may appear in f to introduce a recursive definition. Definitions serve to simplify the program structure, much like definitions in functional programming.

It is important to note that an Orc expression may publish multiple value (or none at all) unlike a functional expression that produces a single value. Unlike functional composition, compositions of Orc expressions are given by sequential composition, of the form $f >x> g$, which may create multiple instances of g, each of which may also publish multiple values.

2.3 An Evaluation of the Design

We draw the reader's attention to some the features that have been omitted. As described earlier, data types and their operators are not part of the Orc language. There is no conditional; we rely upon a fundamental Orc site: $if(b)$, where b is a boolean, returns a *signal* (a unitary data value) if b is **true** and does not respond (remains silent) if b is **false**. This site can be used to effect a computation in which responses received from sites can be filtered. There is no looping mechanism, whose effect we simulate using recursion. There is no specific communication mechanism. For instance, if f and g need to communicate in $f \mid g$, they will have to call a common site into which f, for example, may write and g may read; see the example below. The site may implement a channel, shared memory or rendezvous-based communication. Also absent are notions such as processes, fork-join or synchronization [10]. A fork-join calls two sites M and N in parallel and publishes a pair of their values after they both complete their executions.

$$(let(u, v) <u< M) <v< N, \text{ where } let\ (x) \text{ publishes the value of } x$$

Real time is not a built-in feature. We postulate a fundamental site, called $Rtimer()$, which returns a signal after a specified amount of time. Time-out of expression f is simulated by running a call to $Rtimer()$ as a concurrent expression and aborting f if $Rtimer()$ responds first: $let(x) <x< (f \gg Rtimer(10))$. A similar mechanism is used to interrupt execution of an expression.

One of the more interesting aspects of sites is that a site response may be a site. This allows us to discover services in the internet by calling a site that returns the service site. More interestingly, we can employ such a higher-order site to create a channel that becomes local to an expression. Thus,

$$BuildChannel() >c> f >x> (c.put(x) \mid c.get >x> g)$$

allows f and g to communicate over a local channel c, which was created by calling $BuildChannel()$. This example also illustrates the use of compound sites

containing multiple entry points, in the style of object-oriented programming. In this case the channel site has two entry points, *put* and *get*. This mechanism is completely general so that a variety of communication primitives can be added to a program by having similar builder-sites. We have shown that such sites can be used to create logical clocks, thus supporting discrete event simulation [7].

It is easy to see that we need some combinator to allow concurrent execution of programs (which is given by symmetric composition), for sequencing computations (given by sequential composition) and abortion or interrupt (given by asymmetric composition). However, the form of the combinators, especially for asymmetric composition, was far from obvious, and it is not clear that these are the only concerns in concurrent programming. A considerable amount of experimentation helped in determining these forms, applying the combinators to solve typical practical problems in concurrency and deriving their algebraic properties. There is a number of small programming exercises in [10,7]. The formal semantics of Orc is treated in [12], and its simplicity emboldens us. The combinators also satisfy a number of algebraic identities, which we describe next.

2.4 Algebraic Identities

Some of these identities are inspired by Kleene algebra (the algebra of regular expressions), for which we treat $>x>$ as a generalization of concatenation and $\mid$ as alternation. There is no counterpart of Kleene star in Orc, its effect being subsumed by recursive definitions, and there is no counterpart of $<x<$ in Kleene algebra.

Notation. We write $x \notin f$, for variable x and expression f, to denote that x is not a free variable in f. Below M is an arbitrary site and $\mathbf{0}$ represents a site that never responds. *Signal* is a site that responds immediately with a signal.

$$
\begin{array}{lcll}
f \mid \mathbf{0} & = & f & \\
f \mid g & = & g \mid f & \\
(f \mid g) \mid h & = & f \mid (g \mid h) & \\
\mathbf{0} >x> f & = & \mathbf{0} & \\
(f >x> g) >y> h & = & f >x> (g >y> h), & \text{if } x \notin h \\
\mathit{Signal} \gg f & = & f & \\
f >x> \mathit{let}(x) & = & f & \\
(f \mid g) >x> h & = & (f >x> h) \mid (g >x> h) & \\
(f <x< g) <y< h & = & f <x< (g <y< h), & \text{if } y \notin f \\
(f \mid g) <x< h & = & (f <x< h) \mid g, & \text{if } x \notin g \\
(f >y> g) <x< h & = & (f <x< h) >y> g, & \text{if } x \notin g \\
((f <x< g) <y< h) & = & ((f <y< h) <x< g), & \text{if } y \notin g \text{ and } x \notin h \\
(f <x< g) & = & f \mid (\mathbf{0} <x< g), & \text{if } x \notin f \\
\mathbf{0} <x< M & = & M \gg \mathbf{0} &
\end{array}
$$

It is worth noting that the following laws of Kleene algebra do not hold in Orc: (1) idempotence of $\mid$, i.e., $f \mid f \neq f$, (2) right zero, i.e., $f >x> \mathbf{0} \neq \mathbf{0}$, (3)

left distributivity of $>x>$ over $\mid$, i.e., $f >x> (g \mid h) \neq (f >x> g) \mid (f >x> h)$. The last non-identity is also a non-identity in π-calculus.

2.5 Concluding Remarks

We have expressed our views on what is required for concurrent system design and how Orc meets some of these challenges. Some of our early studies of system design using Orc have been encouraging. We are still trying to understand how transactions may be defined in Orc. Also, work is underway to exploit the platforms designed to facilitate communications among web services, such as SOAP[2], WSDL[3] and UDDI[11], and the XML standard [4] for parameter passing.

References

1. Benton, N., Cardelli, L., Fournet, C.: Modern concurrency abstractions for C#. TOPLAS 26(5), 769–804 (2004)
2. Box, D., EhneBuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielson, H.F., Thatte, S., Winer, D.: Simple object access protocol 1.1, http://www.w3.org/TR/SOAP
3. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web services description language 1.1, http://www.w3.org/TR/wsdl
4. Main page for World Wide Web Consortium (W3C) XML activity and information (2001), http://www.w3.org/XML/
5. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, San Francisco (1993)
6. Hoare, C.: Communicating Sequential Processes. Prentice Hall International, Englewood Cliffs (1984)
7. Kitchin, D., Powell, E., Misra, J.: Simulation using orchestration. In: Proceedings of AMAST (2008)
8. Milner, R.: Communication and Concurrency. In: Hoare, C.A.R. (ed.) International Series in Computer Science. Prentice-Hall, Englewood Cliffs (1989)
9. Milner, R.: Communicating and Mobile Systems: the π-Calculus. Cambridge University Press, Cambridge (1999)
10. Misra, J., Cook, W.R.: Computation orchestration: A basis for wide-area computing. Journal of Software and Systems Modeling (May 2006), http://dx.doi.org/10.1007/s10270-006-0012-1
11. W. site on UDDI, http://www.uddi.org/
12. Wehrman, I., Kitchin, D., Cook, W.R., Misra, J.: A timed semantics of orc. Theoretical Computer Science 402(2-3), 234–248 (2008)

Extending Formal Methods for Software-Intensive Systems

Graeme Smith

School of Information Technology and Electrical Engineering
The University of Queensland, Australia

Abstract. Formal methods have proven beneficial in the industrial development of software-intensive systems; not in replacing traditional engineering methods, but in complementing them. They provide means of checking for ambiguities and inconsistencies in requirements, as well as verifying safety and liveness properties, and the correctness of designs.

As complexity increases, the formal methods employed need to deal with a number of concerns. Primarily they need to be able to model a diverse range of software and hardware components. Ideally, they should also be capable of supporting requirement changes allowing 'ideal' functional specifications to be transformed to reflect actual implementations. Additionally, they should support the introduction of architectural design into functional specifications; including designs involving complex dynamic architectures.

This paper proposes one approach to deal with these concerns. The approach builds on and combines three separate areas of research on integrating formal methods, formal requirements development and formal design derivation. Developing more general theories and techniques that can be applied across a wide range of formal notations remains a significant research challenge.

1 Introduction

Software-intensive systems are those where software components are embedded in a predominantly non-software environment. This environment may comprise electronic and mechanical hardware devices, sensors measuring physical phenomenon, and even people. They include highly decentralised systems such as those used for telecommunications and web services, as well as embedded systems such as fly-by-wire aircraft, heart pacemakers and process control systems of chemical and electrical power plants.

Engineering such heterogeneous and decentralised systems is challenging. Furthermore, such systems are often mission-critical or safety-critical in the sense that their failure could result in massive costs, or in harm or loss of life. Experience has shown that applying formal methods alongside standard engineering approaches to the development of these systems can be extremely beneficial, e.g., [9,36]. They can be used to find ambiguities and inconsistencies in complex sets of requirements, they can be used to verify essential safety and liveness properties, and they can be used to verify that complex designs meet requirements.

M. Wirsing et al. (Eds.): Software-Intensive Systems, LNCS 5380, pp. 146–161, 2008.

To formally model the diverse components of such systems, however, it is desirable that more than one notation be utilised. For hardware components and sensors, we would like a notation supporting real-time and continuous variables. For software components, we would like support for the representation and manipulation of complex data. We also need a means for composing components so that they can operate and interact concurrently. Few formalisms, if any, provide direct support for all of these features.

For components which interface with the real-world, we would also like to be able to start with an 'ideal' specification which only approximates the final implementation. If we were specifying an embedded controller, for example, the abstract specification should not need to account for timing delays and quantisation errors. This can obscure the desired (as opposed to implemented) functionality of the system. It can force the specifier to be distracted by details of the physical implementation and complicate formal analysis. Hence, a means of incrementally modifying specifications to better reflect reality is highly desirable.

Similarly, details of the final implementation architecture of a highly decentralised system may obscure essential system functionality and complicate reasoning. Additionally, the architecture may not be able to be predicted until further into the development process. Hence, there is also a need to allow architectural design to be incrementally added to formal specifications.

Importantly, these specification changes need to be supported without losing the benefits of reasoning at the more abstract level of the initial functional specification. Developing general theories and techniques, applicable to a wide range of formal notations, to address these issues is a significant research challenge. In this paper, we make a first step towards this greater goal by proposing an approach to handling these issues based on a specific formal notation, Real-Time Object-Z [34]. This notation is an integration of the object-oriented, state-based formalism Object-Z [28] and the real-time Timed Interval Calculus (TIC) [11]. Both of the base formalisms have previously found successful application in the development of software-intensive systems in industry: Object-Z in the telecommunications sector [9], and TIC in the aerospace sector [36].

Real-Time Object-Z is presented along with an overview of the field of integrating formal methods in Section 2. In Section 3, we present an existing approach to allowing requirements changes to TIC specifications and discuss its use with Real-Time Object-Z. In Section 4, we discuss recent work on introducing architectural design into Object-Z, and show how it can be adapted to Real-Time Object-Z. We conclude in Section 5 with a discussion of further challenges arising from the types of software-intensive systems likely to be developed and deployed in the coming decades.

2 Integrating Formal Methods

To model software-intensive systems it is inevitable that we will need to use more than one formal notation. A single system may include such disparate aspects

as complex data, concurrency and mobility, and real-time and probability. Few specification languages, if any, are suited to modelling all such aspects. Even if we were only interested in developing the software components, it is essential to also model their environment to get their specifications right [14].

This has led to a growing interest over the past decade in the integration of formalisms. The earliest of this work in the late 1990's focused on the integration of modular state-based formalisms such as Object-Z [28] and B [1] with process algebras such as CSP [15] and CCS [24]. A number of approaches were developed including Object-Z/CSP [32], CSP-OZ [12] and CSP‖B [27]. The motivation behind these approaches was to allow the specification of concurrent components with more complex state than allowable with process algebras alone. More recent work along these lines has seen the integration of Object-Z and B with the π-calculus [25] to support systems with mobility as well as concurrency [35,17].

The need to also model real-time constraints in many applications additionally led to the integration of state-based formalisms with real-time process algebras. For example, TCOZ [18] integrates Object-Z with Timed CSP [5]. To better support the specification of software-intensive systems, TCOZ also allows continuous variables [19].

State-based notations have also been directly integrated with formalisms supporting continuous variables. Most notably, Object-Z has been combined with the Timed Interval Calculus (TIC) [11] by Smith and Hayes [34], and CSP-OZ has been integrated with the Duration Calculus [4] by Hoenicke and Olderog [16]. The former combination referred to as Real-Time Object-Z is discussed in detail below.

2.1 Real-Time Object-Z

Real-Time Object-Z [34] is an integration of Object-Z and TIC aimed at specifying systems in which both complex data structures and continuous real-time variables play a role. Components are specified using Object-Z's class construct extended with TIC predicates describing the component's environmental assumptions and effects. These predicates constrain the behaviour of the Object-Z class and define its interactions with its continuous environment. For example, given that $\mathbb{T}$ denotes absolute time (in seconds), the following expresses that a variable[1] $v : \mathbb{T} \to \mathbb{R}$ becomes equal to a continuous and differentiable (denoted by the function symbol $\rightsquigarrow$ [10]) variable $u : \mathbb{T} \rightsquigarrow \mathbb{R}$ within 0.1 seconds whenever $u > 10$.

$$\langle u > 10 \rangle \subseteq \langle \delta = 0.1 \rangle \,;\, \langle v = u \rangle$$

The brackets $\langle\ \rangle$ are used to specify a set of time intervals. The left-hand side of the above predicate denotes the set of all time intervals where, for all times t in the intervals, $u(t)$ is greater than 10. The right-hand side of the above expression comprises two sets of intervals. The first uses the reserved symbol

[1] Variables in TIC are total functions mapping times to the value the variable assumes at those times.

δ which denotes the duration of an interval. Hence, this set contains all those intervals with duration 0.1 seconds. The second set denotes all intervals in which (for all times in the intervals) v equals u. It is combined with the first set of intervals using the concatenation operator ';'. This operator forms a set of intervals by joining intervals from one set to those of another whenever their end points meet. (One endpoint must be closed and the other open [11]). Hence, the right-hand side of the predicate specifies all those intervals where after 0.1 seconds, v equals u. The entire predicate, therefore, states (using $\subseteq$) that all intervals where u is greater than 10, are also intervals where, after 0.1 seconds, v equals u.

The structure of a Real-Time Object-Z class is as follows where the *assumption* and *effect* are TIC predicates denoting environmental assumptions, and the effect of the specified system when these assumptions hold, respectively.

ClassName

constant definitions
state schema
initial state schema
operations

assumption

effect

The *constant definitions* include continuous environmental variables that act as inputs to the specified system. These variables are declared as functions over all time consistent with the TIC style. They are distinguished by having their names end with the symbol "?".

The *state schema* includes the declaration of both local variables, and environmental variables that act as outputs to the specified system. The latter are distinguished by having their names end with the symbol "!".

The *initial state schema* includes a predicate which constrains the initial values of the variables declared in the state schema. The *operations* detail possible state transitions. Each operation has a list of the variables which it may change, referred to as a *delta-list*, and a predicate constraining the relationship between the values of the variables before and after the operation. Those after the operation are denoted using primes, i.e., x' is the value of variable x after an operation.

As an example of Real-Time Object-Z, consider specifying a speedometer (based on that specified in [19]) which calculates the speed of a vehicle by detecting the rotation of one of its wheels: the speed is calculated by dividing the wheel circumference by the time taken for a single rotation. We assume a maximum speed of 60 metres per second (216 km/hr).

$MaxSpeed == 60$ −metres per second

The speed output by the speedometer is a natural number between 0 and *MaxSpeed*.

$$Speed == 0 \,..\, MaxSpeed \qquad \text{−metres per second}$$

The complete specification of the speedometer is provided by the following class.

Speedometer

$wheel\_circum == 3$ −metres

$wheel\_angle? : \mathbb{T} \rightsquigarrow \mathbb{R}$

$last\_calculation : \mathbb{T}$
$speed! : Speed$

INIT

$last\_calculation < \tau - 2 * wheel\_circum$
$speed! = 0$

CalculateSpeed

$\Delta(last\_calculation, speed!)$

$wheel\_angle?(\tau) \bmod 2\pi = 0$
$\forall\, t : (\tau \,..\, \tau'] \bullet wheel\_angle?(t) \bmod 2\pi \neq 0$
$last\_calculation' = \tau$
$speed!' = wheel\_circum / (\tau - last\_calculation) \pm 0.5$

$\langle | \underline{s}\ wheel\_angle? | \leqslant 2\pi * MaxSpeed / wheel\_circum \rangle = \langle \text{true} \rangle$

$\langle wheel\_angle? \bmod 2\pi = 0 \rangle \,;\, \langle wheel\_angle? \bmod 2\pi \neq 0 \rangle \subseteq$
$\quad \langle \text{true} \rangle \,;\, \langle CalculateSpeed \rangle \,;\, \langle \text{true} \rangle$

The speedometer's environment includes a continuous variable (*wheel_angle*?) representing the angle of the wheel in radians from some fixed position. The speedometer calculates the speed (*speed*!) from the wheel angle, which implicitly records the number of whole revolutions of the wheel, and the wheel circumference (*wheel_circum*). To do this it keeps track of the time of the last speed calculation in a state variable *last_calculation*.

Initially, this variable is set to a time more than $2 * wheel\_circum$ seconds before the current time τ. This ensures that the first speed calculation, when the wheel starts rotating, will be zero (since the calculated speed is a natural number with units metres per second and a wheel rotation time of more than $2 * wheel\_circum$ corresponds to a speed of less than 0.5 metres per second). Ensuring the first speed calculation is zero is necessary because the wheel may not undergo a full rotation before it occurs.

The operation *CalculateSpeed* calculates the speed to the nearest natural number based on the wheel circumference and the time since the last calculation.

Its delta-list includes both *last_calculation*, which it sets to the current time τ at the start of the operation, and *speed*!. It is enabled each time the wheel passes the point corresponding to a multiple of 2π radians. The first two predicates of the operation ensure that the wheel angle mod 2π is 0 only for the first time instant of the operation. This prevents the wheel completing an entire rotation before *CalculateSpeed* has finished executing. (Note that intervals of real numbers can be specified using combinations of the brackets [ ] for closed intervals and ( ) for open intervals.)

The latter constraint is feasible since the class's assumption predicate limits the rate of change of *wheel_angle*? ($\underline{s}\ v$ denotes the derivative of a differentiable variable v [10]). This assumption also ensures that the speed calculated by the final predicate of *CalculateSpeed* is less than or equal to *MaxSpeed*. (Note that $\langle \text{true} \rangle$ denotes the set of all possible intervals.)

To ensure that *CalculateSpeed* occurs every time the wheel passes the point corresponding to 0 radians, the class's effect predicate states that *CalculateSpeed* occurs in a sub-interval of any interval where the wheel angle mod 2π is 0, and then becomes non-zero.

Semantics and Refinement. Specifications in TIC are also usually expressed in terms of assumption and effect predicates (following the approach of the timed refinement calculus [20]). The semantics of such specifications is given in terms of the set A of behaviours over all time satisfying the assumption, and the set E of behaviours over all time satisfying the effect, $\sigma(A, E)$. The basis for the integration of Object-Z and TIC in Real-Time Object-Z is a mapping of the existing Object-Z semantics to a similar set B of behaviours over all time [34]. This allows the semantics of a class to be given as $\sigma(A, E \cap B)$ where A and E are the class's assumption and effect predicates respectively.

Within the integration, both the Object-Z and TIC parts of a specification are given their standard meaning. This allows them to be considered separately for the purposes of reasoning and refinement. Refinement [26,6] refers to the transformation of a specification to another while preserving externally observable properties. It is used to relate abstract specifications to more concrete specifications reflecting an implementation.

Importantly, in Real-Time Object-Z whenever one part of the specification is refined according to the techniques available for its formalism (Object-Z or TIC), the entire specification is also refined [34]. This follows from the fact that a specification with semantics $\sigma(A, E)$ is refined by another with semantics $\sigma(A', E')$ when the assumption of the latter is the same as or weaker than that of the former (i.e., $A \subseteq A'$), and the effect of the latter is the same as or stronger than that of the former (i.e., $E' \subseteq E$) [20]. Hence, a refinement of the TIC part of a Real-Time Object-Z specification will result in a refinement of the entire specification. Furthermore, a refinement of the Object-Z part will not allow additional behaviours (i.e., $B' \subseteq B$) [7], and hence will also result in a refinement of the entire specification.

The notion of refinement is central to the effective use of formal methods. We want our initial specifications to be void of implementation detail for reasons

of clarity and ease of analysis. However, we also want to show that the actual implementation is consistent with such a specification. This can be achieved by showing that another specification which includes the details of the implementation is a refinement of the abstract specification. Limitations with this approach, and proposed solutions are discussed in the following sections.

3 Requirements Change

Refinement is a linear process predicated on the existence of a perfect abstract specification which anticipates all of the necessary characteristics of the final implementation. For software-intensive systems, this would mean accounting for low-level details such as timing delays and quantisation errors. Consider, for example, the TIC predicate relating variables u and v of Section 2. This specification is, in fact, unrealisable. We could not implement a system which sets a variable v to be equal to a continuous real-world variable u. The reading of variable u (by a sensor) would necessarily include some finite error. Hence, the best we could hope for is that $v = u \pm e$ for some small e.

However, including such errors in an initial abstract specification not only makes the specification more difficult to understand, it also complicates reasoning about the specification. For example, what was a deterministic assignment to v becomes non-deterministic. Therefore, we would prefer to specify systems 'ideally' without referring to such physical characteristics of the implementation.

This raises the question then of how we relate such 'ideal' specifications to implementations. This is not possible using refinement. To address this issue, complementary approaches to refinement, such as *realisation* [29,33] and *retrenchment* [2], have been proposed. We discuss the former below.

3.1 Realisation

Realisation, like refinement, is a stepwise-development approach. Realisation steps allow a specification to be changed, either by introducing new requirements or by modifying existing ones. Importantly, the realisation steps transform not only the specification, but also any formal properties already proven about that specification. This allows formal reasoning carried out on the original specification to be reused after the specification has been modified, avoiding the need to repeat formal proofs.

A complete set of realisation rules for TIC is presented in [29] and applied to a non-trivial case study in [33]. As an example, consider the rule which allows an assumption to be added to a specification. Such a transformation could not occur via refinement which only allows the weakening of the overall assumption. The rule states how properties of the specification are transformed when it is applied.

> If P is a property of a specification with assumption A and effect E then $B \Rightarrow P$ is a property of the specification with assumption $A \wedge B$ and effect E.

For example, assume the predicate involving variables u and v in Section 2 and modified to include an error e as above is the effect predicate of a specification with no assumption, i.e., the assumption predicate is true. From this specification we are able to prove that $\forall t : \mathbb{T} \bullet u(t) > 10 \Rightarrow v(t + 0.1) = u(t + 0.1) \pm e$. It is not possible to implement this specification, however, unless the rate of change of u is restricted. Hence, we need to introduce an assumption $\forall t : \mathbb{T} \bullet |\underline{s}\ u(t)| \leqslant R$ where R is the maximum rate of change. Our property is consequently changed to $(\forall t : \mathbb{T} \bullet |\underline{s}\ u(t)| \leqslant R) \Rightarrow (\forall t : \mathbb{T} \bullet u(t) > 10 \Rightarrow v(t + 0.1) = u(t + 0.1) \pm e)$. The other realisation rules detailed in [29] work similarly.

Given the semantics of a Real-Time Object-Z class described in Section 2.1, adding an assumption to the TIC part of a class, adds the same assumption to the entire class. Hence, the rule above can also be used with Real-Time Object-Z. The other realisation rules involve replacing environmental variables with new variables that are explicitly related to the original ones allowing the introduction of timing delays and errors (see [29] for details). For example, consider the rule for modifying an environmental input variable.

> If P is a property of a specification with assumption A, effect E and an input u then $(\forall u \bullet F \Rightarrow A) \Rightarrow (\exists u \bullet F \wedge P)$ is a property of the specification where u is replaced by a fresh variable related to u by F.

In contrast to the rule for adding assumptions, applying this rule to the TIC part of the class will not apply it to the entire class. The Object-Z part has been written expecting the original 'ideal' input, not the new input. To use such a rule with Real-Time Object-Z, however, we can retain the original variable in the Object-Z part of the class (as a local variable) and relate it to the new environmental variable by including F as part of the class's effect predicate.

Hence, we can apply the existing realisation rules to the TIC part of a Real-Time Object-Z class in order to move towards a more realistic specification. Note, however, that some real-time constraints occur in the Object-Z part of a class. A complete realisation approach would therefore need a way of also transforming these constraints.

4 Architectural Design

Just as time delays and quantisation errors may obscure desired functionality and complicate reasoning, so too may elements of the implementation architecture of a software-intensive system. For example, certain components such as system clocks, while necessary in the final implementation, unnecessarily complicate a purely functional specification. Also, in some cases, the architecture may only be decided upon later in the development process than when the specification is required. For these reasons, it is desirable to be able to introduce architectural design into an initial specification.

To do this, refinement may be used as the design should not change the high-level requirements. To support such an approach, Smith [30] has integrated Real-Time Object-Z with the process algebra CSP [15]. This integration is based

on that of Object-Z/CSP [32], and provides a mapping between the semantics of Real-Time Object-Z classes and CSP processes. This allows such classes to be combined using CSP operators; in particular, the parallel operator of CSP is used so that the occurrence of an operation in one class is synchronised with that in another with the same name. The semantic mapping allows such synchronising operations to merely overlap in time and not necessarily have common start or end times (see [30] for details).

Given this integration with CSP, we could develop a means of introducing architecture into specifications via refinement (such an approach for Object-Z/CSP is presented in [8]). However, the architectures we can readily specify with CSP operators are limited. In particular, CSP has not been designed to represent dynamic architectures where the number of components and their interaction with each other change over time (although there is limited work on this topic [38]). This could perhaps be overcome by using a process algebra such as the π-calculus [25] rather than CSP. In this section, however, we consider an alternative approach based on using the constructs of Object-Z itself.

4.1 Specification Refactoring

Object-Z supports modelling using object-oriented concepts such as inheritance and object instantiation. Smith [31] shows that this makes it well suited to the specification of mobile systems. Specifically, the creation and passing of object references can readily model architectures which can change dynamically.

In recent work, McComb and Smith [21,23,22] have investigated an idea akin to refactoring in object-oriented programming to introduce and remove architectural structure within Object-Z specifications. We discuss the adaptation of this work here for Real-Time Object-Z.

Assume we want to implement the speedometer of Section 2 in terms of the three components in Figure 1. That is, a wheel sensor component sends pulses to the main speedometer component whenever the wheel completes a rotation. The latter component counts pulses from a clock component in order to calculate the speed.

To derive a specification reflecting this architecture, we begin by refining the class *Speedometer* of Section 2 to separate the concerns of the three components.

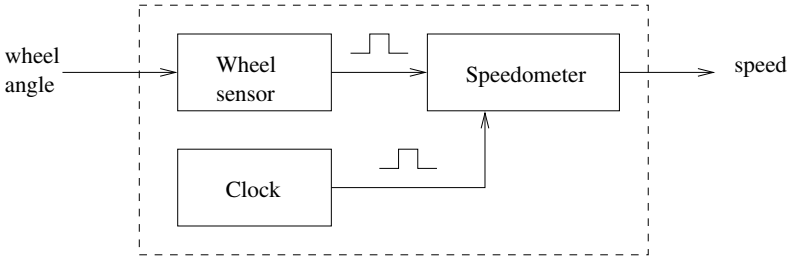


Fig. 1. Speedometer design

To do this, we add a new constant denoting the clock period ($freq = 1$ MHz), replace the variable *last_calculation* by a variable which counts the clock pulses (*count*), and add a variable denoting the time of the last clock pulse (*last_clock*). Since the original class calculated the speed to the nearest 0.5 metres/second, the class below is a refinement provided that the clock frequency is sufficient to allow at least $MaxSpeed/0.5$ clock pulses in the minimum possible wheel rotation time $wheel\_circum/MaxSpeed$. The frequency of 1MHz is sufficient to ensure this.

Speedometer

$wheel\_circum == 3$ −metres
$freq == 10^6$ −hertz

$wheel\_angle? : \mathbb{T} \rightsquigarrow \mathbb{R}$

$count : \mathbb{N}$
$last\_clock : \mathbb{T}$
$speed! : Speed$

Init

$count > 2 * wheel\_circum * freq$
$last\_clock = \tau$
$speed! = 0$

ClockPulse

$\Delta(last\_clock)$

$\tau = last\_clock - 1/freq$
$last\_clock' = \tau$
$\tau' < \tau + 1/freq$

IncCount

$\Delta(count)$

$count' = count + 1$

$Count \mathrel{\widehat=} ClockPulse \land IncCount$

WheelPulse

$wheel\_angle?(\tau) \bmod 2\pi = 0$
$\forall t : (\tau \ldots \tau'] \bullet wheel\_angle?(t) \bmod 2\pi \neq 0$

NewSpeed

$\Delta(count, speed!)$

$count' = 0$
$speed!' = wheel\_circum/(count/freq)) \pm 0.5$

$CalculateSpeed \mathrel{\widehat=} WheelPulse \land NewSpeed$

$\langle | \underline{s}\ wheel\_angle? | \leqslant 2\pi * MaxSpeed/wheel\_circum \rangle = \langle \text{true} \rangle$

$\langle wheel\_angle? \bmod 2\pi = 0 \rangle \,;\, \langle wheel\_angle? \bmod 2\pi \neq 0 \rangle \subseteq$
$\quad \langle \text{true} \rangle \,;\, \langle WheelPulse \rangle \,;\, \langle \text{true} \rangle$
$\langle \delta \geqslant 1/freq \rangle \subseteq \langle true \rangle \,;\, \langle ClockPulse \rangle \,;\, \langle true \rangle$

The operation *CalculateSpeed* is redefined as the conjunction of two new operations *WheelPulse* and *NewSpeed*, and new operations *ClockPulse* and *IncCount* are conjoined to form a final new operation *Count*. (Note that we adopt the view of [21] which allows widening a class's interface under refinement.) The predicate

of *ClockPulse* together with a new effect predicate ensures that the operation occurs every 10^{-6} seconds.

We then refactor the class into the following three classes.

Speedometer

$sensor : WheelSensor$
$clock : Clock$
$count : \mathbb{N}$
$speed! : Speed$

INIT

$count > 2 * sensor.wheel\_circum * clock.freq$
$clock.INIT$

IncCount

$\Delta(count)$

$count' = count + 1$

$Count \mathrel{\widehat=} clock.ClockPulse \land IncCount$

NewSpeed

$\Delta(count, speed!)$

$count' = 0$
$speed!' = sensor.wheel\_circum/(count/clock.freq)) \pm 0.5$

$CalculateSpeed \mathrel{\widehat=} sensor.WheelPulse \land NewSpeed$

$true$

$true$

WheelSensor

$wheel\_circum == 3$ −metres

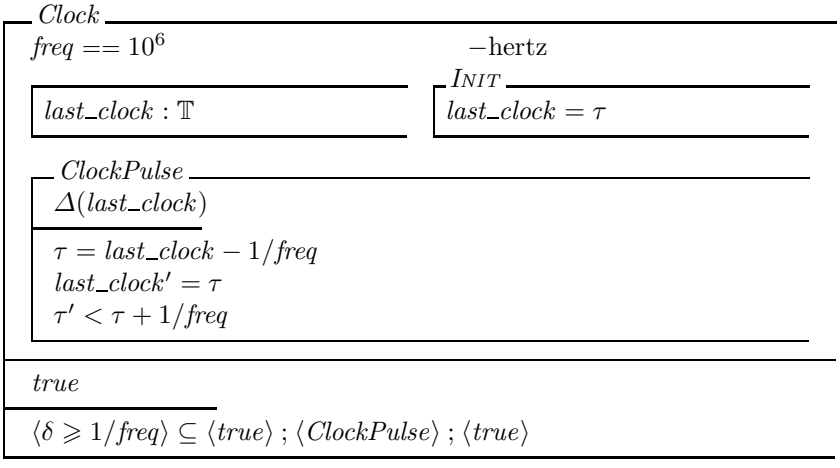
$wheel\_angle? : \mathbb{T} \rightsquigarrow \mathbb{R}$

WheelPulse

$wheel\_angle?(\tau) \bmod 2\pi = 0$
$\forall\, t : (\tau \ldots \tau'] \bullet wheel\_angle?(t) \bmod 2\pi \neq 0$

$\langle | \underline{s}\ wheel\_angle? | \leqslant 2\pi * MaxSpeed/wheel\_circum \rangle = \langle \text{true} \rangle$

$\langle wheel\_angle? \bmod 2\pi = 0 \rangle \,;\, \langle wheel\_angle? \bmod 2\pi \neq 0 \rangle \subseteq$
$\langle \text{true} \rangle \,;\, \langle WheelPulse \rangle \,;\, \langle \text{true} \rangle$



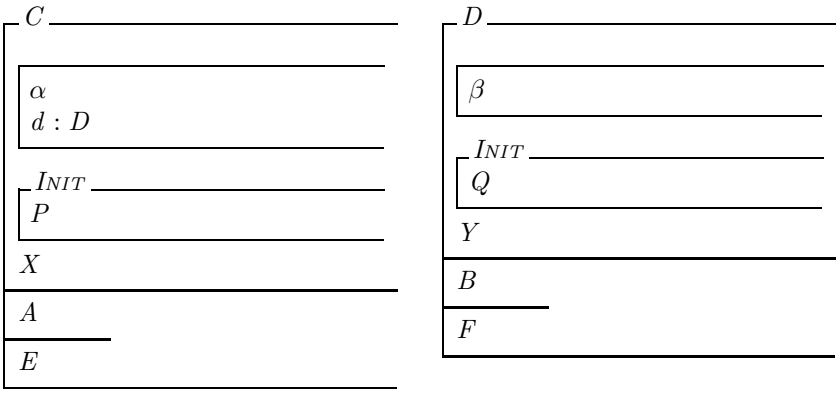
For this refactoring to preserve the observable properties of the initial specification, we require that the new *Speedometer* class is a refinement of the original one. In order to prove this, we first need to provide a semantics of object instantiation in Real-Time Object-Z. This semantics needs to extend that for Object-Z to (a) ensure that operations on an object synchronise as required with those in the class where the object is declared, and (b) take into account the TIC assumption and effect predicates in an object's class.

The first extension is required so that, for example, the *IncCount* operation in the final *Speedometer* class above occurs whenever the *ClockPulse* operation of *clock* occurs. Simply conjoining the operations does not ensure this as the τ variables of the operations are local to their respective classes.

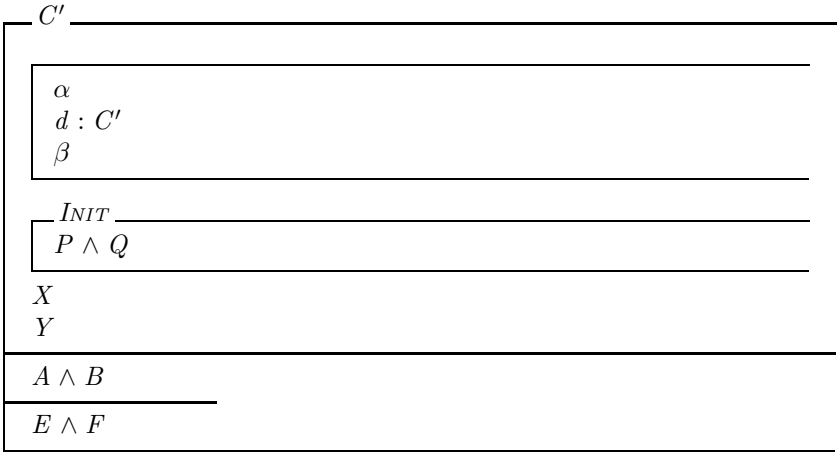
One approach to ensure synchronisation is to equate the τ variable of an object with that of the class in which it is declared. This is overly restrictive, however. In our example, it would force the conjoined operations from *Speedometer* and *Clock* to have identical start and end times. In general, synchronising operations may simply overlap.

An alternative approach is to introduce an implicit output variable $t! : \mathbb{T}$ to every operation and an implicit predicate $\tau \leqslant t! \leqslant \tau'$. Since conjoined operations need to agree on output variables such as $t!$, they would need to have at least one time in common between their start and end times. That is, they would need to overlap in time (without necessarily sharing start and end times) as desired.

To tackle the second extension, we can make use of the fact that, as in other object-oriented languages, self referencing of classes is allowed in Object-Z [28]. This enables us to define a single class which is equivalent to a system of interacting classes. The idea, inspired by McComb's work on refactoring in Object-Z [22], is to create one class which has all the features of every class in the system, and so can act as the class of every object. The approach is illustrated for a class C with a single reference to an object of class D below (where α and β are non-intersecting sets of declarations of variables and constants, and X and Y are non-intersecting sets of operations).



is (in the absence of object aliasing) semantically equivalent to



While the object referenced by variable d is of class C', rather than D, in C', class C' has all the features that are dereferenced in the predicate P and schemas in the set X, i.e., all the features of the original class D. Furthermore, the assumption B and effect F of the original class D are applied to these features.

To handle more general architectures, we can replace the declaration $d : C'$ in C' above, with $d : Id \pfun C'$ to represent an arbitrary number of objects each indexed via a value from a set Id. The corresponding declaration of objects in class C would be $d : Id \pfun D$. Since the operations in X could modify the objects which make up the range of d, dynamic architectures could be readily modelled.

5 Future Directions

This paper has examined three areas of research in formal methods that are relevant to software-intensive systems: the use of multiple formalisms, the

incremental introduction of physical characteristics to functional specifications, and the incremental introduction of architectural design. It has also considered issues in combining the ideas from these areas with the goal of providing support for requirements analysis and design assurance in the traditional engineering of such systems.

The types of software-intensive systems envisaged in the coming decades provide additional challenges to those we have addressed in this paper. Firstly, the decreasing cost and size of computational devices will lead to increasingly distributed systems. Massive numbers of components will interact and potentially move about networks in order to jointly perform complex tasks. Object-oriented formalisms, such as Object-Z, seem well suited to this challenge since they allow models comprising arbitrarily large collections of dynamically linked objects. The main issue will be the derivation of complex designs consisting of massive numbers of objects to fulfil high-level requirements. The refactoring approach presented in Section 4 could provide a basis for this. The original approach for Object-Z by McComb has been proven complete in the sense that the derivation of any valid architecture is possible [22]. To make derivation of large-scale architectures practical, however, would require the additional development of high-level design 'tactics' driving the application of large sequences of simple refactoring rules.

Additionally, the components of future systems will need to become more adaptive, able to autonomously change their behaviour based on their perceived environment. This suggests the systems will be multi-agent systems [37]. The combination of the modularity and expressive logic of Object-Z makes it again suited to the challenge of formalising such systems. In fact, Object-Z has already been independently proposed as a modelling language for multi-agent systems [13]. The concept of realisation discussed in Section 3 may also be useful for such systems. It could be used to determine how the properties of a component change as its behaviour changes. This would enable the specification of complex adaptable behaviour in terms of a number of specifications of simpler behaviour.

Finally, new technologies will be introduced into future systems. These technologies may make use of radically different computing paradigms and materials (e.g., nanotechnology). It is inevitable that some of these will require new formalisms and methods of reasoning. Hence there will be a need for further integration of notations. The semantic mapping approach to integration discussed in Section 2 allows the straightforward introduction of new notations as it requires no changes to the syntax or semantics of the notations involved. Additionally, further investigation into the integration of formal methods and standard modelling and analysis techniques from engineering and science, in particular numerical methods, is required [3].

Acknowledgements. The author would like to acknowledge ARC Discovery Grant DP0558408, and Kirsten Winter for reading an earlier draft of this paper.

References

1. Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Banach, R., Poppleton, M., Jeske, C., Stepney, S.: Engineering and theoretical underpinnings of retrenchment. Science of Computer Programming 67(2-3), 301–329 (2007)
3. Boulton, R., Gottliebsen, H., Hardy, R., Kelsey, T., Martin, U.: Design verification for control engineering. In: Boiten, E., Derrick, J., Smith, G. (eds.) IFM 2004. LNCS, vol. 2999, pp. 21–35. Springer, Heidelberg (2004)
4. Chaochen, Z., Hoare, C.A.R., Ravn, A.P.: A calculus of durations. Information Processing Letters 40, 269–271 (1991)
5. Davies, J., Schneider, S.: A brief history of Timed CSP. Theoretical Computer Science 138(2), 243–271 (1995)
6. de Roever, W.-P., Engelhardt, K.: Data Refinement: Model-Oriented Proof Methods and their Comparison. Cambridge University Press, Cambridge (1998)
7. Derrick, J., Boiten, E.: Refinement in Z and Object-Z, Foundations and Advanced Applications. Springer, Heidelberg (2001)
8. Derrick, J., Smith, G.: Structural refinement of systems specified in Object-Z and CSP. Formal Aspects of Computing 15(1), 1–27 (2003)
9. Duke, R., Rose, G., Smith, G.: Transferring formal techniques to industry: A case study. In: Quemada, J., Mañas, J., Vazquez, E. (eds.) Formal Description Techniques (FORTE 1990), pp. 279–286. North-Holland, Amsterdam (1990)
10. Fidge, C.J., Hayes, I.J., Mahony, B.P.: Defining differentiation and integration in Z. In: Staples, J., Hinchey, M.G., Liu, S. (eds.) International Conference on Formal Engineering Methods (ICFEM 1998), pp. 64–73. IEEE Computer Society Press, Los Alamitos (1998)
11. Fidge, C.J., Hayes, I.J., Martin, A.P., Wabenhorst, A.K.: A set-theoretic model for real-time specification and reasoning. In: Jeuring, J. (ed.) MPC 1998. LNCS, vol. 1422, pp. 188–206. Springer, Heidelberg (1998)
12. Fischer, C., Wehrheim, H.: Failure-divergence semantics as a formal basis for an object-oriented integrated formal method. Bulletin of the EATCS 71, 92–101 (2000)
13. Gruer, P., Hilaire, V., Koukam, A., Cetnarowicz, K.: A formal framework for multi-agent systems analysis and design. Expert System Applications 23(4), 349–355 (2002)
14. Hayes, I.J., Jackson, M., Jones, C.B.: Determining the specification of a control system from that of its environment. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 154–169. Springer, Heidelberg (2003)
15. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
16. Hoenicke, J., Olderog, E.-R.: CSP-OZ-DC: a combination of specification techniques for processes, data and time. Nordic Journal of Computing 9(4), 301–334 (2002)
17. Karkinsky, D., Schneider, S., Treharne, H.: Combining mobility with state. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 373–392. Springer, Heidelberg (2007)
18. Mahony, B., Dong, J.S.: Timed Communicating Object Z. IEEE Transactions on Software Engineering 26(2), 150–177 (2000)
19. Mahony, B.P., Dong, J.S.: Sensors and actuators in TCOZ. In: Wing, J., Woodcock, J.C.P., Davies, J. (eds.) FM 1999. LNCS, vol. 1709, pp. 1166–1185. Springer, Heidelberg (1999)

20. Mahony, B.P., Hayes, I.J.: A case-study in timed refinement: A mine pump. IEEE Transactions on Software Engineering 18(9), 817–826 (1992)
21. McComb, T.: Refactoring Object-Z specifications. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. LNCS, vol. 2984, pp. 69–83. Springer, Heidelberg (2004)
22. McComb, T.: Formal Derivation of Object-Oriented Designs. PhD thesis, The University of Queensland (2007)
23. McComb, T., Smith, G.: Architectural design in Object-Z. In: Strooper, P. (ed.) Australian Software Engineering Conference (ASWEC 2004), pp. 77–86. IEEE Computer Society Press, Los Alamitos (2004)
24. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
25. Milner, R.: Communicating and Mobile Systems: The π-Calculus. Cambridge University Press, Cambridge (1999)
26. Morgan, C.: Programming from Specifications. Prentice-Hall, Englewood Cliffs (1990)
27. Schneider, S., Treharne, H.: Communicating B machines. In: Bert, D., Bowen, J., Henson, M., Robinson, K. (eds.) B 2002 and ZB 2002. LNCS, vol. 2272, pp. 416–435. Springer, Heidelberg (2002)
28. Smith, G.: The Object-Z Specification Language. Advances in Formal Methods. Kluwer Academic Publishers, Dordrecht (2000)
29. Smith, G.: Stepwise development from ideal specifications. In: Edwards, J. (ed.) Australasian Computer Science Conference (ACSC 2000). Australian Computer Science Communications, vol. 22, pp. 227–233. IEEE Computer Society Press, Los Alamitos (2000)
30. Smith, G.: An integration of Real-Time Object-Z and CSP for specifying concurrent real-time systems. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335, pp. 267–285. Springer, Heidelberg (2002)
31. Smith, G.: A formal framework for modelling and analysing mobile systems. In: Australasian Computer Science Conference (ASCS 2004), pp. 193–202. Australian Computer Society (2004)
32. Smith, G., Derrick, J.: Specification, refinement and verification of concurrent systems - an integration of Object-Z and CSP. Formal Methods in Systems Design 18, 249–284 (2001)
33. Smith, G., Fidge, C.: Incremental development of real-time requirements: The light control case study. Journal of Universal Computer Science 6(7), 704–730 (2000)
34. Smith, G., Hayes, I.J.: An introduction to Real-Time Object-Z. Formal Aspects of Computing 13(2), 128–141 (2002)
35. Taguchi, K., Dong, J.S., Ciobanu, G.: Relating pi-calculus to Object-Z. In: International Conference on Engineering of Complex Computer Systems (ICECCS 2004), pp. 97–106. IEEE Computer Society, Los Alamitos (2004)
36. Wildman, L.: Requirements reformulation using formal specification: a case study. In: Lakos, C., Esser, R., Bristensen, L.M., Billington, J. (eds.) Workshop on the use of Formal Methods in Defence Systems, pp. 75–83. Australian Computer Society (2002)
37. Wooldridge, M.: An Introduction to MultiAgent Systems. John Wiley & Sons, Chichester (2002)
38. Zakiuddin, I., Goldsmith, M., Whittaker, P., Gardiner, P.: A methodology for model-checking ad-hoc networks. In: Ball, T., Rajamani, S. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 181–196. Springer, Heidelberg (2003)

Ensemble Engineering and Emergence

Hu Jun*, Zhiming Liu, G.M. Reed, and J.W. Sanders

International Institute for Software Technology,
United Nations University, P. O. Box 3058 Macao, SAR China
{lzm,mike,jeff}@iist.unu.edu
Also: College of Computer and Communication,
Hunan University, Changsha, China
hujun_111@hnu.cn

Abstract. The complex systems lying at the heart of ensemble engineering exhibit emergent behaviour: behaviour that is not explicitly derived from the functional description of the ensemble components at the level of abstraction at which they are provided. Emergent behaviour *can* be understood by expanding the description of the components to refine their functional behaviour; but that is infeasible in specifying ensembles of realistic size (although it is the main implementation method) since it amounts to consideration of an entire implementation. This position paper suggests an alternative. 'Emergence' is clarified using levels of abstraction and a method proposed for specifying ensembles by augmenting the functional behaviour of its components with a system-wide 'emergence predicate' accounting for emergence. Examples are given to indicate how conformance to such a specification can be established. Finally an approach is suggested to Ensemble Engineering, the relevant elaboration of Software Engineering. On the way, the example is considered of an ensemble composed of artificial agents and a case made that there emergence can helpfully be viewed as ethics in the absence of free will.

1 Introduction

The large complex systems that currently exist, either by explicit design or by accretion, have been called *ensembles* by the Interlink Working Group on software intensive systems and new computing paradigms (see the Interim Management Report [36]). Examples include the power grid, the internet and large systems of agents (including swarms *etc.*). Naturally there is healthy debate about the characteristic properties of an ensemble, amongst which are included: a massive number of components and behaviour that is open and adaptive (as a result of being situated in the real world) and emergent and statistical (rather than being able predominantly to be addressed at the individual level).

To assist the process of classification, the Interlink group has divided ensembles into two kinds, physical and societal. Examples of the former are: very large

* Supported by the National Natural Science Foundation of China under Grant No. 60773208.

M. Wirsing et al. (Eds.): Software-Intensive Systems, LNCS 5380, pp. 162–178, 2008.

adaptive sensor or robot systems; systems composed of programmable molecules; advanced manufacturing systems; the internet. Examples of the latter are: large traffic systems; swarm or colony behaviour; systems of interacting businesses; the stockmarket. Typically ensembles are systems of systems that were not necessarily designed to be composed but adapt, reconfigure and self-organise. Common to all should be a theory of ensembles, and ensemble engineering.

Ensembles are engineering products, too recent for appropriate supporting theories to have arisen. Such theories would provide the right abstractions for specifying, developing, reasoning about and programming ensembles. Without them, the state of the art will remain at the engineering level; with them there is the prospect of controlling and thus further exploiting ensembles. For standard systems, that is the domain of Software Engineering and its foundation, Formal Methods.

The group ended its second workshop having made considerable progress in demarking areas of interest for future medium and longer-term work, but with some uncertainty concerning emergent behaviour. Clearly, it felt, emergence is a unifying theme across the spectrum of examples. Yet if an ensemble exhibits behaviours not predictable from those of its components, what part can Formal Methods play in ensemble engineering? After all, the utility of Formal Methods lies in the specification of systems and the verification of implementations or designs against their specifications. Does such reductionism mean that these systems lie outside the scope of Formal Methods? And what might ensemble engineering look like?

Those are the topics addressed in this position paper (of which [15] is a preliminary version). Its purpose is: to clarify the place of emergence in the types of system quoted above (Section 2); to consider typical examples and be guided by them (Section 3); and to suggest an agenda for laying a foundation of ensemble engineering (Section 5). On the way it is observed that in the special case of ensembles of agents, the emergent behaviour can profitably be thought of as the result of ethical protocols of the agents, imposed at a societal level. That leads, if the agents are artificial, to an interesting theory of ethics weaker than the usual theory (for sentient agents, based on free will); it is discussed in Section 4.

2 Ensembles

A definition of 'ensemble' based on any of the quantitative features like those mentioned in the previous section (size, as measured say by number of components, and so on) is not going to support a very interesting theory,[1] regardless of the number of its exemplars. This section proposes instead to study the important place of emergence in such systems by abstracting all other properties and *defining* an ensemble to be a system exhibiting emergent behaviour. That way any conclusions apply to all the examples above.

[1] 'Theory' here is interpreted in the formal sense of comprising only the consequences of the definition. Thus interest focuses on a definition strong enough to support an interesting and appropriate theory whilst being weak enough to apply to the range of desired examples.

But then a definition of 'emergence' is required. 'Emergence' is an established term about which the working group expressed rough consensus only after considerable discussion—presumably reflecting the divergence of interpretations in the literature. Again, and for the same reason as with the definition of 'ensemble', this paper takes a minimalist view and restricts the definition of 'emergence' to just 'system behaviour not derivable (at the stated level of abstraction) from the behaviour of its components alone'. This suppresses any 'element of surprise' sometimes discussed in the philosophy of emergence [8]. The details are as follows.

2.1 Levels of Abstraction

The term 'emergence' was coined by Lewes [21] in 1875 since when it has enjoyed a lively and varied existence. Perhaps that is why a comprehensive history is difficult to find.[2] Since the twentieth century, it has been associated with complex systems. Indeed it seems that each twist of science or philosophy imbues the term with its own flavour.

The present contribution is no exception. It arises from Formal Methods and its rigorous description at a prescribed 'level of abstraction' (LoA for short; plural: LoAs). But the basis of our approach is far from new. According to Pepper over 80 years ago,

> The theory of emergence involves three propositions: (1) that there are levels of existence ... (2) that there are marks which distinguish these levels from one another ... (3) that it is impossible to deduce marks of a higher level from those of a lower level ...
>
> S. C. Pepper, [26].

We use the familiar notation of Formal Methods to interpret Pepper's 'levels of existence' as 'LoAs' in a way which is entirely conceptual, so that the levels need not correspond to any naive idea of observation. But first it is necessary to recall the notion of LoA, on which Formal Methods is based.

A formal description of a system consists, regardless of the notation used, of a predicate whose free variables are the system observables, and which therefore determine the LoA of the description. For analogue, differentiable, systems the observables are rates of change of system parameters and the predicate corresponds to the solution of a differential equation which yields the system state at any given time. In that case the standard concepts of Differential Analysis complement those of Computer Science to facilitate description and analysis (see Section 3.2).

But if the system is discrete, so that the observables assume only finitely-many values, then the predicate can be expressed in terms of system state, input and output.[3] There the notations and concepts of Formal Methods are required to structure (particularly, large) descriptions and provide them with semantics. For hybrid systems a combination of both those styles is to be expected.

[2] Sketch histories of emergence are given in the Stanford Encyclopedia of Philosophy [31] and Wikipedia [34].

[3] Of course there is a trade-off between state and input-output history; hence the need for 'can be expressed'.

In either the discrete or non-discrete case, the result is a predicate whose free variables determine the LoA of the system description. To liberate our treatment from any particular Formal Method, we make the following definitions.

Definition (LoA). An *observable* is a typed variable together with an informal interpretation of what it represents. (For example $l : \mathbb{R}^3$ might be an observable representing location of an object in Euclidean space.) An observable is *discrete* iff its type is finite.

A *level of abstraction, LoA,* consists of a finite nonempty set of observables. A LoA is *discrete* iff each of its observables is discrete; it is called *analogue* iff each of its observables is not discrete; and otherwise (if some observables are discrete and some are not) it is called *hybrid*.

A *behaviour* at a given LoA is a predicate whose free variables are the observables of that LoA; values of the observables making the predicate true correspond to the specified 'system behaviour'. (For example the predicate ensuring that the location $l = (x, y, z)$ lies in the first quadrant, namely $x \geq 0 \wedge y \geq 0 \wedge z = 0$, describes a system behaviour of the object mentioned above.)

Suppose a system is captured at two LoAs as follows. At level A it has behaviour p_A. Level C is defined to extend level A by fresh observables from some type B, $C = A \cup B$, and to have behaviour $p_C = p_C(a, b)$. We say that the former is *abstract* and the latter *concrete* iff p_A is weaker than p_C with the new observables abstracted:

$$(\exists\, b : B \cdot p_C(a, b)) \;\Rightarrow\; p_A.$$

(For example augmenting the observable l above with an observable $t : \mathbb{R}$ for time and constraining the location to lie on the x axis, yields a concrete observation/behaviour.) Sometimes the abstract level is called *high* and the concrete *low*. □

For more elaborate examples of those concepts and an extension to the more involved relationship between abstract and concrete that pertains when the latter is a data representation of the former, we refer to [11].

2.2 Emergence

Now emergence is simply explained: the LoA at which the system is observed lies at a lower level than that at which the components are specified. In the case of a flock of birds, for example, the components are the birds specified unilaterally at a LoA sufficient for just that purpose; but the flock is observed at a LoA consisting of the previous one augmented by further observables relating to flock behaviour (for example, 'location of a bird in the flock' makes sense only at the flock level—although a distributed implementation might enforce it by providing each bird with a (bird dependent) 'strategy' for its location within the flock). More detailed behaviour is now able to be observed at the (lower) flock level: the required emergent behaviour situates birds correctly.

For a system to exhibit emergence, not all behaviour possible at the lower level may satisfy the desired criterion for emergence. For example there are 'potential

flocks' that position birds incorrectly, and so do not conform to the required (emergent) definition of flock. Otherwise, the low-level observables just introduced would not serve to discriminate any behaviours and all low-level behaviour would appear emergent. But then all low-level behaviour would be anticipated in the high-level behaviour, contrary to the desired meaning of 'emergence'. This apparently subtle point forms an essential part of the definition.

Definition (Emergence). Suppose a system is described at two LoAs, A and C, as above. The system is said to exhibit *emergent behaviour*, or simply *emergence*, as expressed by behaviour (*i.e.* predicate) em_C of C, if $em_C \Rightarrow p_C$ and em_C describes some behaviours not determined by p_A:

$$\exists\, a : A \cdot \exists\, b, b' : B \cdot em_C(a, b) \wedge \neg em_C(a, b')\,. \tag{1}$$

Because it describes emergent behaviour, the predicate em_C is called the *emergence predicate* of the pair of system descriptions. □

The setting for the definition of emergence uses the simplest common relationship between A and C: the latter is a restriction (by the emergence predicate) of an extension (by the fresh variables) of the former. More complicated settings are possible (for example, if C is a restriction of a data refinement of A).

Because the observables b are fresh, an observation a, b cannot be made at the abstract level. But to ensure that it cannot be trivially inferred from an abstract observation, condition (1) is imposed. The following contrived but simple example is designed to clarify that point.

Example. A system is designed to have abstract behaviours consisting of $a : \mathbb{B}$ (where $\mathbb{B}$ denotes the type of Booleans) and concrete behaviours having type $c = (a, b) : \mathbb{B} \times \mathbb{B}$. Three putative emergence predicates are defined:

$$\begin{aligned} em_0(a, b) &\mathrel{\widehat{=}} \neg a \\ em_1(a, b) &\mathrel{\widehat{=}} \mathit{true} \\ em_2(a, b) &\mathrel{\widehat{=}} \neg b \vee a\,. \end{aligned}$$

Neither em_0 nor em_1 can be considered emergent because neither uses the augmenting (fresh) variable b to specialise behaviours. In either case the behaviour could be modelled, by suitable interpretation, at just the abstract level. That is not true of em_2, just because it satisfies condition (1). □

Here is a less contrived example, to be elaborated in Section 3.2.

Example. In the case of the flock of birds, $a : A$ consists of the states of the birds, independent of each other but parameterised to make them individual; so A is a (large) product space with one component for each bird. The type B includes a component for 'location within the flock'. The predicate em includes a conjunct placing each bird in its correct (though perhaps approximate, depending on the exact nature of the description) location. On the other hand, introduction of an observable corresponding to 'flock sleep' does not produce emergent behaviour if it occurs exactly when all individual birds sleep. □

In some Formal Methods, like alphabetised process algebra, the two descriptions em_C and its abstraction $\exists\, b : B \cdot em_C$ are deemed incomparable, exactly because the types of their free variables differ. Such theories are therefore not obvious candidates as the basis for a theory in which passage from components to the whole ensemble (or *vice versa*) is required. In others like data refinement, translation *via* a simulation relation is required before the two levels of behaviour can be directly compared.

2.3 Ensembles

Having clarified the definitions of 'LoA' and 'emergence', the definition of 'ensemble' is now straightforward.

Definition (Ensemble). A system forms an *ensemble* iff it has emergent behaviour: its components are described abstractly whilst its system-wide behaviour is described as the combination of the abstract components augmented by variables and, in terms of them, an emergence predicate. □

Thus a system forms an ensemble if its behaviour is not derivable (at the stated LoA) from that of its components alone. How is an ensemble to be described? In Section 3 a mild variant of the notation Object-Z [10] is used because it allows the components to be described in a modularised manner, and an emergence predicate to be added.

It is important to appreciate that the definition depends on LoA: a system may exhibit emergence and so form an ensemble when described at one pair of LoAs but not at another. This property of the definition is crucial for ensemble engineering, as will appear. At a more fundamental level, it resolves the tension between emergence and reductionism.

2.4 Reductionism

The concept of emergence can be viewed as providing a 'patch' to fill a gap in the systematic reduction of the behaviour of the whole to that of its parts: with reductionism [35]. But how is that to be reconciled with Formal Methods, which after all relies on the gap-free decomposition of a complex system into formalised components. If that methodology does not capture all system behaviour then it is seriously flawed. This section provides a reconciliation.

The tension between emergence and reductionism is long standing and has been extremely well documented since Descartes. Much of the confusion can be clarified by making explicit the LoA of a description (as described in the previous section) [11]. As will be seen from the examples in Section 3 and, as already pre-empted, at the specification level there is typically insufficient state in the components to account for emergent behaviour of the ensemble. To expand component state would be tantamount to describing an implementation; but then what was emergent in the specification would no longer be emergent in the implementation. What is emergent at one level of abstraction (for us, the

abstract level of specification) may not be at another (for us, the implementation level).

Formal Methods supports the top-down incremental derivation of a system from its specification. At intermediate stages the resulting construct, usually called a design, is part specification and partly executable (already code). It is to be expected, then, that derivation will yield intermediate LoAs in which the ensemble's functionality is being captured in an efficiently executable manner—as usual—but also that the emergent behaviour is gradually being accounted for.

A design thus represents a step towards ensuring that the specified components are fit for ensuring the emergent behaviour. At the specification level (exhibiting emergence) the components by themselves (*i.e.* before incorporating the emergence predicate) are not fit; at the implementation level (since no emergence remains) they are fit; and in between they are being modified to make them fit. Throughout this paper 'fit' refers to just that conformance.

2.5 Related Approaches

Fromm [13] argues that the generality of the concept of emergence makes any definition unlikely and so instead proposes a taxonomy. Whilst examples are useful, so too is standard Mathematics for framing general concepts. That is the approach taken here.

Whilst emergence has been seen here as the explicit structuring of phenomena at one LoA in terms of those at another, there is a more extreme position in the study of Mind according to which, for example,

> ... human level intelligence is too complex and little understood to be correctly decomposed into the right subspecies at the moment and that even if we knew the right subspecies we still wouldn't know the right interfaces between them. Furthermore, we will never understand how to decompose human intelligence until we've had a lot of practice with a simpler level intelligence. R. A. Brooks, [1].

There is, of course, considerable support for this position concerning the all-encompassing notion of intelligence. But the last quoted sentence might be seen as suggestive that simpler forms of intelligence and more restricted LoA be studied first. The hope would then be that a hierarchy of incremental LoAs be used to understand the more detailed behaviour—an approach with which Formal Methods has substantial experience. Damper [8] also discusses reductionism and emergence from the perspective of LoAs. Although we have found that LoAs clarify much of the Philosophical discussion, the fundamental question remains of whether or not there exist ensembles (like Mind) and emergent behaviour (like consciousness) that is not reducible. We take no position on the question. The techniques provided in this paper are designed for the ensembles arising in Computer Science.

In the general setting of complex systems Gell-Mann [14] has suggested that the study of such phenomena be called *plectics*. He introduces 'granularity', which is

conveniently formalised using LoAs. Examples from Artificial Life have played a formative part in our work. For an older summary we refer to Cariani [4].

Ryan [29] makes the important point that, in our terms, when applied to open systems (which ensembles typically are) the concrete LoA may capture components in the environment of the abstract LoA. Although the definition he gives appears to allow it, he debars statistical phenomena from being emergent. That makes his formalism inappropriate for our use here (see Sections 3.1 and 3.2). Chen *et al.* [5] use transition systems (apparently with deterministic transitions) to introduce a classification of emergence, based on Ryan's definition, motivated by systems expressed in process algebra.

For a recent summary of much of the related work that we have no space to review here, we refer to the survey by Deguet *et al.* [9].

3 Examples

The emergent behaviour of an ensemble with a large number of components may be (partly) statistical in nature. That is something with which 'traditional' Software Engineering has little experience in spite of the availability of probabilistic [24] and societal [25] algorithms from Computer Science. So the purpose of the first example, a fair coin, is to consider in as simple a setting as possible the essence of statistical emergence, stripped of the attendant functionality that would make a realistic ensemble more complex. It is to be expected that the statistical 'ingredient' must, by the standards of Statistics, be trivial. The purpose of the example is thus to clarify: how does a statistical property emerge, how is it to be specified (using formal methods) and how is it to be implemented and conformance checked?

The second example, a flock of birds, sketches a dynamically changing ensemble seen from the viewpoint of emergence. A standard from Artificial Life, we again abstract detail to concentrate on position and velocity of each bird in the flock; the points being made—for instance, concerning distributed versus centralised control in a design that conforms to the specification of an ensemble—do not lie in the detail. Interest centres on the manner in which the distributed implementation conforms to emergence specified and how they are expressed in a manner accessible to ensemble engineering.

3.1 Coin Tossing

By abstracting almost all the functionality in an agent-based ensemble, we are led to the following example of a simple ensemble exhibiting statistical behaviour consisting of (a large number of) tosses of a coin. The abstracted components of the ensemble are identical: each is a single coin toss. The ensemble, however, consists of the (large number of) tosses, and the emergent behaviour is the bias—in this case zero—of the coin. Thus there is no way to infer the emergent behaviour from any collection of single components.

In the first specification, the ensemble is countably infinite, permitting expression of the usual criterion that two events 0 and 1 (representing *heads* and

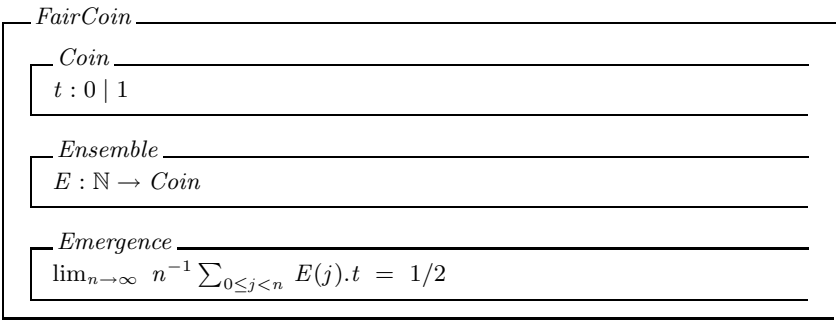


Fig. 1. Specification of the 'probabilistic' fair coin

tails) to be equally probable. In (approximiately) the notation of Object-Z [10], emergent behaviour may be described as if it were liveness of a state-based system. See Figure 1. The system is named *FairCoin*. It has three local constituents: a component named *FairCoin.Coin* whose observable t is either 0 or 1; a countable collection of such components, called *FairCoin.Ensemble*; and an emergence predicate *FairCoin.Emergence*. The first is used simply in order to define the second, whilst the third is conjoined with the second to yield the system behaviours.

Though statistically standard, that infinite ensemble is of mere theoretical interest. A more realistic ensemble contains only a finite number of (unordered) tosses. Then the limit is replaced by some agreed statistical approximation: the fraction of heads in the total number of tosses deviates from half by an amount interpreted as fair at a certain confidence level (based on the binomial distribution and, for a large ensemble, its approximation to the normal distribution using the central limit theorem).

Treating the size N of the ensemble and the degree $c : [0, 1]$ of confidence as parameters (although the latter is typically expressed as a percentage), write $Fair(E, c)$ to mean that the bag E of tosses is fair at the c-confidence level: the difference

$$| N^{-1} \sum_{e \in E} e.t \; - \; 1/2 |$$

lies within the bound dictated by the confidence level c. See Figure 2.

The obvious implementation consists of the iterated toss of a coin which is fair by construction. The program $P \, {}_{\frac{1}{2}}\!\oplus \, Q$ is equally likely to be P or Q [23], and may be implemented directly using a random-number generator. The statistical behaviour of the implementation is now not emergent because it is inferred directly from the iterated components. Again, see Figure 2.

Conformance of *FairCoinImp* to *FairCoinSpec* then follows by standard Statistics (or probability theory, actually, *via* the binomial distribution). Indeed the criterion is the standard one: the behaviour of *FairCoinImp* is a special case of that described by *FairCoinSpec*.

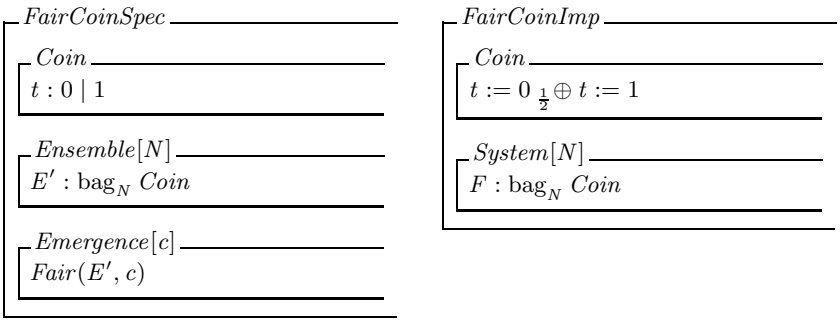


Fig. 2. Specification and implementation of the fair coin

By concentrating on just emergent statistical phenomena, that example has been devoid of interesting state. However it is easy to incorporate a mechanism. For example in interactive science museums can often be found a peg board with pegs configured in such a way to deflect balls into columns that exhibit a binomial (or other) distribution. Such a mechanism is readily modelled as a loop in which, on each iteration, its Boolean variable is assigned 0 (a ball goes to the left) or 1 (to the right) with equal probability (say); and that loop is iterated N times.

The fair coin is representative of a well-known family of examples of emergence in which more interesting distributional behaviour (than binomial) emerges as the result of a more subtle generating mechanism. The family includes Zipf's law, Benford's law and so on. A typically thorough but old treatment of such laws is given by Knuth [18]; see more recently [3,22].

For instance many realistic random variables are lognormally distributed (that is, the logarithm of the variable is normally distributed); that skews them to the right. Examples are heights (usually assumed to be normal), particle sizes, ore deposits, incomes across the population and so on. These days the most convincing explanations for lognormal genesis are founded on the mere process of classifying data. However an elegant mechanism was given by Kolmogorov [19]. An attribute (income, or ore) is dispersed in the population by a series of steps; dispersal is proportional to the power of some constant, but random at each step; so taking logarithms the central limit applies to produce a normal distribution. Such a mechanism can be coded like the peg board above, and produces an example combining both nontrivial state in each component, but statistical emergence in the ensemble.

3.2 Flocks

An ensemble in which more truly dynamic behaviour emerges is that of a flock of birds (swarm of bees, school of fish, herd of quadrupeds, *etc.*); see for example [2]. The components are the individuals, described autonomously (now we abstract statistical behaviour to concentrate on distribution). Such a description suffices to specify the actions of the birds independent of other birds but *en masse*. It

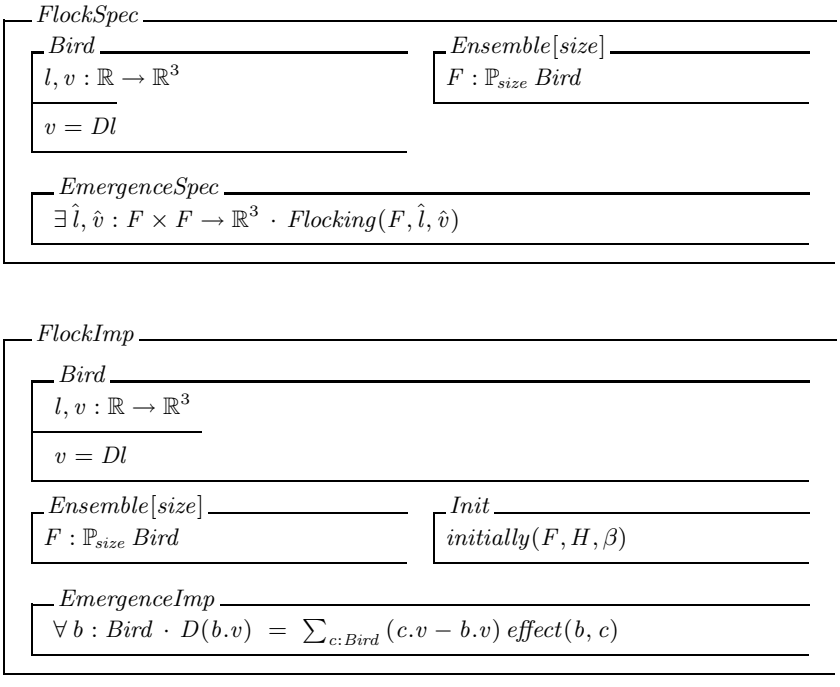


Fig. 3. An (abstracted) specification, *FlockSpec*, and an implementation, *FlockImp*, clarifying the part played by emergence in a flock. At both levels the flock consists of a finite set of birds each of which has, at any time, a location l and velocity v, the latter of which is the derivative of the former. The emergence predicate *Flocking* defines flocking behaviour of the ensemble and is given by Definition (3). In *FlockImp* it is achieved by updating the velocity of each bird according to the average of the difference between its current velocity and that of the other birds, weighted according to the function $effect : Bird \times Bird \to \mathbb{R}$ given by Definition (2). The predicate *initially* represents a condition on its arguments for the convergence to hold in *EmergenceSpec*.

does not account for behaviour of the ensemble as a whole, like flocking, which is thus emergent.

The ensemble is specified in outline as for the previous example; see Figure 3. The specification of the unilateral individuals is given first, with some parameters to account for differences between individuals. Then a collection of individuals, the colony, is defined (a set suffices if individuals are uniquely determined by their parameters). Finally emergence is described. Now it is more subtle, since the behaviour of the colony may result in dynamic reconfigurability: a flock may temporarily divide when confronted by a predator, or other obstacle, and afterwards recombine. Techniques must be developed to describe such reconfigurability elegantly and in a structured manner (but so far little seems to have been done *c.f.* [30]).

In this example, the emergence predicate accounts for the flocking behaviour of individuals. The idea of determining the motion of an individual by averaging

that of its neighbours appears first (in this context) to be due to Reynolds [28]. His model, boids, is based on a heuristic in which 'collision avoidance' has higher priority than 'matching the velocity of neighbours' which in turn has higher priority than 'staying close to neighbours'. A more detailed quantitative model, in a more general but less heuristic setting, was simulated by Vicsek *et al.* [33]. That model was treated analytically by Jadbabai *et al.* [17], who produced convergence results in both discrete and continuous time; adverse local effects (like collision or noise) were not considered there, although leadership of the flock was included.

The most comprehensive analytical results to date are due to Cucker and Smale [6,7] whose treatment extends that of Jadbabai to three dimensions, including noise, for both discrete and continuous time. For contrast with the previous example, we consider the continuous case. Suppose that each bird b has attributes location $l(t) : \mathbb{R}^3$ and velocity $v(t) : \mathbb{R}^3$ at any time $t : \mathbb{R}$. In the flock, an individual's velocity is incremented by a weighted average of the difference between its velocity and those of other birds (the weighting can be instantiated to allow only neighbours to influence that update). In [7] bird interaction is determined by a function $\mathit{effect} : \mathit{Bird} \times \mathit{Bird} \to \mathbb{R}$ giving the effect of its first argument on its second. The symmetric instantiation

$$\mathit{effect}(b, c) \;\widehat{=}\; H(1+ \parallel b.l - c.l \parallel^2)^{-\beta} \tag{2}$$

is shown to be sufficient for flocking, for constants $H > 0$, $\beta \geq 0$ and a simple condition relating those constants to the initial values $(b.l)_0$ and $(b.v)_0$ for the location and velocity, respectively, of each bird b.

Flocking is defined by the condition that both inter-bird distances and velocities converge: for a finite set F of birds b with time-dependent location and velocity observables $b.l, b.v : \mathbb{R} \to \mathbb{R}^3$ there are (unique) time-independent inter-bird location and velocity functions $\hat{l}, \hat{v} : F \times F \to \mathbb{R}^3$ for which this predicate holds:

$$\mathit{Flocking}(F, \hat{l}, \hat{v}) \;\widehat{=}\; \forall\, b, c : F \cdot \begin{pmatrix} \|(b.l)(t) - (c.l)(t) - \hat{l}(b, c)\| \to 0 \\ \|(b.v)(t) - (c.v)(t) - \hat{v}(b, c)\| \to 0 \end{pmatrix} \tag{3}$$

where the norm is the usual Euclidian norm and convergence is with time tending to infinity.

If D denotes differentiation with respect to time then, for all $b : F$, the differential equations

$$\begin{array}{rcl} D(b.l) & = & b.v \\ D(b.v) & = & -\sum_{c:\mathit{Bird}} \mathit{effect}(b, c)\,(b.v - c.v) \end{array}$$

ensure *Flocking*, (recall that on the right the $b.v - c.v$ is a function of t) assuming simple conditions on the initial state of the flock (*i.e.* on the initial values of $b.l$ and $b.v$) and the constants H and β from Definition (2); see [7], Theorem 1. Writing $\mathit{initially}(F, H, \beta)$ for (the conjunction of) those initialisation conditions, the resulting specification and implementation are given in Figure 3.

An implementation—at least a design for one—can, as in any distributed system, vary in its degree of distribution. At one extreme it is centralised; at the other it is entirely distributed. A distributed design would incorporate extra information in each individual to account for behaviour of the group, perhaps with a relatively small amount of randomness in response to the environment. A centralised design—which would seem less appropriate in this case—includes extra components (omniscience, with access to the state of each individual) to account for the emergent behaviour of the group. Techniques for the description of distributed designs are well developed in Computer Science. The criterion for conformance of a design to a specification is again that each behaviour exhibited by the design is allowed by the specification. But sufficient conditions, respecting the modularisation of the ensemble, must be developed.

4 Emerging Ethics

4.1 Ethics without Free Will?

An important case of a multi-agent ensemble is that in which the agents are artificial. Whilst there is no (mathematical) definition of that term (it is 'agent' which is contentious, not 'artificial'!), it seems to be accepted that an artificial agent must be interactive, autonomous and adaptable [12]. In particular, only certain programs are agents. A typical example is provided by reactive software considered at a level of abstraction at which, typically by employing machine learning and making probabilistic choices, it adapts to interactions with its environment. Reconfigurability may be a particular feature; but anyway as a result of adaptability, the ensemble exhibits emergence (compared with the more abstract view).

It is important that such systems be specified. Otherwise their behaviour as they adapt is unpredictable. But there seems to be almost no experience of that: such systems seem to be considered entirely at the implementation level.

In society, an ensemble composed of sentient agents, such emergence can be seen as the result of either laws or ethical principles. When the agents are human (subject to the usual exceptions involving mental immaturity due either to youth or mental state) and so exhibit free will, the field of Ethics provides normative principles by which that dynamic multi-agent ensemble, society, functions within the desired tolerances. But those principles (like deontologism, consequentialism, utilitarianism *etc.*) rely entirely on the agent possessing free will; and they tend largely to focus on the individual.[4] So in the absence of free will, for example in a multi-agent ensemble composed of artificial agents, an alternative foundation must be provided: new normative principles must be developed which do not depend on the agents possessing free will and which apply squarely to systems.[5]

[4] The ethical platforms of various companies and organisations make interesting reading; they all seem to be strongly influenced by individual (human) ethics, even to the treatment of take-overs.

[5] In so far as laws carry over to the artificial case, they are readily specified as functional properties, to be satisfied like any safety property.

The question is whether or not such principles can be strong enough to support an interesting theory. The answer seems to be positive.[6] Such principles will at first seem strange from the point of view of Ethics, for precisely the reason that they are not founded on free will. In many cases they do not look particularly 'ethical', pertaining instead simply to functionality of the multi-agent ensemble. But their utility is to be measured by the way in which, like the principles of standard Ethics, they enable behaviour of agents (collective behaviour, in the case studied here) to be specified, implemented and analysed. They ensure fitness of agents.

'Ground zero' of such principles for multi-agent systems is the 'principle of distribution' [27] according to which each system action should be as distributed as possible. It finds application in many distributed systems, including those from socio-economics and politics. The case of an individual (artificial) agent is considered in [12]; further examples appear in [32]. Indeed ethical considerations, when interpreted in this suitably abstracted manner not involving free will, play an important part in motivating the designs of many distributed systems [30].

4.2 Emergence and Conformance

Section 3 has demonstrated that it is both appropriate and convenient to specify an ensemble as a conjunction of components augmented by an emergence predicate (each may of course be a conjunction). More generally an ensemble might be naturally expressed in terms of further ensembles. It would be of interest to consider laws that transform an ensemble to some canonical form.

It would also be of interest to consider various kinds of emergence. We have concentrated on statistical (Section 3.1) and limiting spatial/temporal (Section 3.2) behaviour. There might be a temptation, for example, to say what an ensemble 'ought' to do. Indeed it is to be expected that deontic logic will provide an important notation for expressing types of emergence. But it must be stressed that a property is only of use if conformance to it can be established. It is as well to be specific, since new techniques are to be expected in ensemble engineering:

> A specification is a system description against which conformance can be decided.

Such is the case, for instance, for a large family of probabilistic properties based on expectations. It might at first be thought otherwise, on the grounds that any finite behaviour is consistent with a given frequency being attained in the limit. In fact the highly successful theory [23] has been assumed implicitly in the examples in Section 3.

But for deontic logic the position seems to be far from successful. There appears to be no denotational semantics and the only (Kripke) semantics already assumes on the possible worlds a semantic notion of duty. Thus it is of no help

[6] The name 'information ethics' has been given to that weaker theory, and investigated by the Information Ethics Group at Oxford [16].

to say, without further clarification, that an ensemble's emergent (or any other) behaviour is that it *ought* to perform some action.

Having decided a denotational semantics for ensembles exhibiting emergent behaviour, laws and criteria for conformance will be important.

5 Conclusion

The importance of societal engineering and ensemble engineering seems assured. This position paper has considered one aspect of both: the place of emergent phenomena in such systems and the manner in which it can be formalised and implemented. That seems sufficient to justify the following research agenda whose purpose is to clarify the importance of emergence. The result is expected to be a foundation for ensemble engineering in the context of the Interlink programme.

1. *Description.* Developing notation for specifying ensembles and for expressing designs of ensembles. The use of Object-Z here has been a first attempt. What emergence predicates arise (as stastistical distribution and spatio-temporal convergence have here) and how are they best expressed? Can deontic logic be made useful? (And if so, does it have a denotational semantics that can be used for the verification of laws, as has been done for probabilism [23]?) What intermediate designs arise in ensuring that a component agent conforms to emergent behaviour?
2. *Conformance.* Give criteria, and practical sufficient conditions, for one design to conform to a specification or another design. This necessarily includes a semantics for the notations developed in the previous part. For the standard and probabilistic descriptions used here that has already been done *via* Object-Z and the probabilistic guarded command language *pGCL*. But, as a warning: the Software Engineering of probabilistic systems is far from easy because of the interaction between probabilism and nondeterminism. What about other emergence predicates? And the return to fitness of individuals and the system? The semantics makes available sound laws, which justify the extent to which a conjunction of ensembles may be reduced to a single ensemble. That should be investigated.
3. *Case studies.* Consider a range of case studies, representative of realistic ensembles. Important examples include those featured in this proceedings, agent-based ensembles, dynamically reconfigurable ensembles, machine learning, and designs that account for emergence with varying degrees of distribution. In particular the specification of the environmental emergence exhibited by adaptive (machine learning) systems seems both interesting and important.[7] Vital here is the 'self-stability' of such dynamic ensembles to return to their 'stable' system state after perturbations.
4. *Model checking.* Show that some of the case studies conform to their specifications by automated verification of the sufficient conditions established above.

[7] The approach of Section 3 can be applied to show that learning may be treated as emergence; see [15], Section 3.3.

More speculative topics, that are nonetheless of interest, include:

1. *Continuous ensembles?* With a very large number of similar components, is there a place for reasoning as if the ensemble were infinite and then using an approximation for large finite ensembles? That would permit the standard theory of differentiability to be used to reason about the limiting case, and afterwards use a discrete approximation to infer behaviour of the ensemble in hand. If such an approach is of use, what is the place of hybrid ensembles?
2. *Game theory* In 'strategic' ensembles, whose component agents compete for advantage, it is to be expected that the best theories available for describing emergent behaviour are game theoretic. It would be interesting to have a realistic but feasible case study of this kind.
3. *Artificial ethics?* Is it useful to pursue the idea of the ethical responsibility of artifical agents, and to use emergent 'ethical' qualities in specifying them?

Acknowledgements

The authors are grateful to the referees and Graeme Smith for improvements and corrections. They also appreciate the lively and productive discussions surrounding this topic in the Interlink Workshops under Martin Wirsing's deft guidance.

References

1. Brooks, R.A.: Intelligence without representations. Artificial Intelligence 57, 139–159 (1991)
2. http://www.aridolan.com/ad/Alife.html
3. Brown, G., Sanders, J.W.: Lognormal Genesis. Journal of Applied Probability 18(2), 542–547 (1981)
4. Cariani, P.: Emergence and artificial life. In: [20], pp. 775–797 (1991)
5. Chen, C.-C., Nagl, S.B., Clack, C.D.: A calculus for muilti-level emergent behaviours in component-based systems and simulations. In: Proceedings of Emergent Properties in Natural and Artificial Complex Systems (EPNACS 2007), pp. 35–51 (2007), http://www-lih.univ-lehavre.fr/~bertelle/epnacs2007-proceedings/epnacs07proceedings.pdf
6. Cucker, F., Smale, S.: Emergent behaviour in flocks. IEEE Transactions on Automatic Control 52(5), 852–862 (2007)
7. Cucker, F., Smale, S.: On the mathematics of emergence. The Japanese Journal of Mathematics 2, 197–227 (2007)
8. Damper, R.I.: Emergence and levels of abstraction. Editorial for the special edition on Emergent properties of complex systems. International Journal of Systems Science 31(7), 811–818 (2000)
9. Deguet, J., Demazeau, Y., Magnin, L.: Elements about the emergence issue: a survey of emergence definitions. ComPlexUs 3, 24–31 (2006)
10. Duke, R., Rose, G.: Formal Object-Oriented Specification Using Object-Z. Macmillan Press, Basingstoke (2000)
11. Floridi, L., Sanders, J.W.: The method of abstraction. In: Negrotti, M. (ed.) Yearbook of the Artificial. Models in Contemporary Sciences, vol. 2. Peter Lang (2004)

12. Floridi, L., Sanders, J.W.: On the morality of artificial agents. Minds and Machines 14(3), 349–379 (2004)
13. Fromm, J.: Types and forms of emergence. Nonlinear Sciences, abstract (June 13, 2005), arxiv.org/pdf/nlin.AO/0506028
14. Gell-Mann, M.: The Quark and the Jaguar. W. H. Freeman and Company, New York (1994)
15. Jun, H., Liu, Z., Reed, G.M., Sanders, J.W.: Position paper: Ensemble engineering and emergence (and ethics?). UNU-IIST Technical report 390 (December 2007), http://www.iist.unu.edu
16. Information Ethics Group, University of Oxford, http://web.comlab.ox.ac.uk/oucl/research/areas/ieg
17. Jadbabaie, A., Lin, J., Morse, A.: Coordination of groups of mobile autonomous agents using nearest neighbor rules. IEEE Transactions on Automatic Control 48, 988–1001 (2003)
18. Knuth, D.E.: The Art of Computer Programming, 2nd edn. Seminumerical Algorithms, vol. 2. Addison-Wesley, Reading (1981)
19. Kolmogorov, A.N.: C.R. Dokl. Acad. Sci. URSS 30, 301–305 (1941)
20. Langton, C.G., Taylor, C., Farmer, J.D., Rasmussen, S. (eds.): Artificial Life II. Santa Fe Institute Studies in the Sciences of Complexity, Proceedings 10. Addison-Wesley, Redwood City (1992)
21. Lewes, G.H.: Problems of Life and Mind. First series, vol. 2. Trübner & Co., London (1875)
22. Li, W.: Random texts exhibit Zipf's-law-like word frequency distribution. IEEE Transactions on Information Theory 38(6), 1842–1845 (1992)
23. McIver, A.K., Morgan, C.C.: Abstraction, Refinement and Proof for Probabilistic Systems. Springer Monographs in Computer Science (2005)
24. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press, Cambridge (1995)
25. Pauly, M.: Logic for Social Software. PhD. thesis, CWI Amsterdam (2001)
26. Pepper, S.C.: Emergence. Journal of Philosophy 23, 241–245 (1926)
27. Reed, G.M., Sanders, J.W.: The principle of distribution. Journal of the American Society for Information Science and Technology 59(7), 1134–1142 (2007)
28. Reynolds, C.: Flocks, herds, and schools: a distributed behavioral model. Computer Graphics 21(4), 25–34 (1987)
29. Ryan, A.: Emergence is coupled to scope, not level. Complexity 13, 67–77 (2007)
30. Sanders, J.W., Turilli, M.: Dynamics of Control. In: Theoretical Advances in Software Engineering 2007, TASE 2007, pp. 440–449. IEEE Computer Society, Los Alamitos (2007)
31. The Stanford Encyclopedia of Philosophy, http://plato.stanford.edu/entries/properties-emergent
32. Turilli, M.: Ethical protocols design. Ethics and Information Technology 9(1), 49–62 (2007)
33. Vicsek, T., Czirók, A., Ben-Jacob, E., Cohen, I., Shochet, O.: Novel type of phase transition in a system of self-driven particles. Phys. Rev. Lett. 75(6), 1226–1229 (1995)
34. Wikipedia, http://en.wikipedia.org/wiki/Emergence
35. Wikipedia, http://en.wikipedia.org/wiki/Reductionism
36. Wirsing, M.: (working group leader). InterLink WG 1 Interim Management Report (IMR), WG 1: Software intensive systems and new computing paradigms (June 2007)

Mathematical Support for Ensemble Engineering*

Michael Johnson

Macquarie University, Sydney, Australia
mike@ics.mq.edu.au

Abstract. We study some of the mathematical challenges presented by the need to support ensemble engineering, concentrating on likely contributions from category theory and universal algebra. Particular attention is paid to dealing with missing data, modelling dynamics and interaction, and analysing inconsistencies.

Keywords: Ensemble engineering, category theory, universal algebra, inconsistency analysis.

1 Introduction

This volume, and much of the earlier work of the INTERLINK Working Group 1 (WG1), has advocated *ensemble engineering* as an important new computing paradigm. For a discussion of *ensembles*, the reader is referred to [3] in which ensemble engineering is defined (page 19) as "the science and engineering discipline of complex, integrated ensembles of computing elements ... [and] ways to reliably and predictably model, design, and program them".

The growth in the development of distributed systems, mobile technologies, agent-based systems, multi-processor embedded systems, and the construction of interoperations for legacy systems all contribute engineering techniques that can be useful for ensemble engineering. At the same time, the challenges of very large numbers of nodes, adaptive technologies which blur the boundary between development time and operation time, open environments, and emergent behaviour, will require the development of new mathematical techniques for reliable design.

Some of the mathematics expected to be of value can be predicted already. Probabilistic analyses, differential equations, the modal logic of games, and many aspects of complex systems theory are all relevant. This paper explores the prospects of providing mathematical support for ensemble engineering using one less known mathematical tool — category theoretic universal algebra. After some background material and a brief example of specification via category theoretic universal algebra we deal in turn with the need to develop mathematics for the

* Research partially supported by the Australian Research Council. I thank the anonymous referees for useful comments and the editors for encouraging me to include the survey material in Sections 2 and 3.

M. Wirsing et al. (Eds.): Software-Intensive Systems, LNCS 5380, pp. 179–189, 2008.

dynamics of algebras, proposals for the study and management of the integration of systems, how to deal with limited data at the local level of computational elements, how to deal with limited data during interoperation between computational elements, and a proposal for the analysis of systems in the presence of inconsistencies.

2 Remarks on Category Theory

Some readers of this volume will have little or no experience with the branch of mathematics known as category theory. The remainder of this paper attempts to outline results which follow from category theoretic analysis without including the mathematical details. Of course, there is a risk of "falling between two stools" — those who understand category theory may feel cheated by the missing details, while those with no experience of category theory may wonder what it's all about.

To try to address the first group, the category theorists, I have included precise statements of the category theory involved whenever I can do that with a few words. Those without experience of category theory should just read past the technical terms when they appear.

For the second group I say a few words informally about category theory in this section. Mostly I try to avoid full definitions, talking *about* category theory rather than trying to provide an exposition of a graduate course in a page or two.

Nevertheless, we begin with some detailed definitions: A *category* is a directed (multi-) graph, together with a composition for arrows in the graph defined whenever two arrows meet head to tail (viz $A \longrightarrow B \longrightarrow C$, with the resulting composite a single arrow from A to C). If the two composable arrows are called f and g then the composite is denoted gf (note the order which corresponds to the usual (algebraic) notation for composite functions). The composition operation is required to be associative (so $h(gf) = (hg)f$) and to have identities.

Since in every category composition of arrows is associative, any string of arrows $A \longrightarrow B \longrightarrow \ldots \longrightarrow T$ in a category has a unique composite. Of course, different strings might have the same composite. If another string of arrows $A \longrightarrow C \longrightarrow \ldots \longrightarrow T$ has the same composite the diagram made up of the two strings is said to *commute*. Some examples of commutative triangles and a commutative square appear in Section 4.

Common examples of categories arise from classes of mathematical structures and the arrows, often called *morphisms*, between them. For example, the category of finite sets has as objects all finite sets and as arrows the functions between the sets. Similarly there are categories whose objects are groups and whose arrows are group homomorphisms; topological spaces with continuous maps; graphs with graph homomorphisms; etc. There is even a category whose objects are all "small" categories and whose arrows are the appropriate morphisms for categories — graph morphisms ϕ which respect the composition meaning that $\phi(gf) = \phi(g)\phi(f)$ and ϕ takes identity arrows to identity arrows — called *functors*. (Do not be concerned about "size" issues which cause no difficulties provided one correctly uses classes).

Finite categories are quite common in computer science applications. They are easily presented by giving their finite underlying graph and tabulating the (finitely many) compositions.

Categories were introduced during the 1940s, originally to make precise the notion of *natural transformation* — a kind of morphism between functors. The language of categories has since become widely used in many ares of mathematics, and category theory itself has had an important role in unifying and organising disparate areas of mathematics.

Categories are disarmingly simple — a category is merely a graph together with a composition which is associative and has identities. What is surprising is that such a simple notion can have much "semantic power". A large part of that power comes from the discovery during the 1950s that notions nowadays known as *limits* and *colimits*, including *products*, *coproducts*, *pullbacks* and *pushouts*, can be described solely as properties of arrows within a category. Such properties typically take the form "for all arrows of a certain kind there is a unique arrow of another kind making certain diagrams commute", and because of the initial universal quantifier ("for all") they are known as *universal properties*. Another kind of universal property, *cartesian morphisms*, plays a role in Section 8.

The existence of objects with certain universal properties is sometimes called an *exactness condition*. For example, to say that a category, like the category of finite sets, has all finite products (which it does), is an exactness condition. A category which has all finite limits is called *finitely complete*. Finite completeness is quite a strong exactness condition, but we do sometimes require more, for example the existence of finite coproducts (Section 4). The category of finite sets has all of these exactness conditions and more besides.

3 Categorical Universal Algebra

In the 1960s, F.W. Lawvere discovered and developed categorical universal algebra. Lawvere observed that using finite products and commutative diagrams he could construct a category which encapsulated all of the information required for a particular branch of algebra, say group theory. The category is called the *theory* of a group. Every finite group arises as a finite product preserving functor from the theory to the category of finite sets. Indeed more: The category whose objects are such functors and whose arrows are the natural transformations between them is *equivalent* to the category of finite groups and group homomorphisms.

Similarly other areas of algebra arise correspondingly from other theories. Monoids, semigroups, rings and many other algebraic structures can be treated in exactly the same way. And theorems proved about the categories using category theoretic tools are theorems of universal algebra.

In order to further generalise from fully defined operations (like the product of elements of a group) to partially defined operations (like the composition of arrows in a category) one needs to replace "finite products" with more general limits. For example, a certain pullback can be used to abstractly specify the *composable pairs* of arrows in an abstract category. Thus, one is led to the notion

of a *theory* as a category with certain exactness properties, and algebras as functors from the theory to a category of sets which preserve the exactness properties. Commutative diagrams in the theory still correspond to axioms that the algebras are required to satisfy.

In the 1970s Lawvere and others developed categorical logic, most explicitly in *topos theory*, in which certain universal properties can be used to represent standard logical constructions. Categorical logic is important to us here because category theory can be used to specify systems as well as algebras, and restrictions on those systems are often expressed in logical terms which can, since the 1970s, be rephrased into category theoretic specifications.

To conclude this very brief historical survey we note that in the 1980s people began widely using category theory and universal algebra (although not usually category theoretic universal algebra) for specifications in computer science. In the 1990s the author and others used Lawvere style category theoretic universal algebra to specify information systems while others used category theory for program language semantics or even to introduce programming constructs (eg *monads* in Haskell). Since 2000 the author and his colleague Rosebrugh have been using category theory to study view updates (Section 8), and view updates to engineer system interoperations (for arguably very very small ensembles).

4 System Specification Using Universal Algebra

This section briefly reviews the mathematical foundation the author has used for system specification using category theory. It is based on categorical universal algebra, which is the basis of classic algebraic specification techniques [2]. We assume some familiarity with elementary category theory, as might be obtained in [1], [11] or [13]. For the purposes of this paper we will just outline the basic ideas. A fuller treatment can be found in, for example, [7].

A *theory* is a finitely complete category, frequently with other exactness properties (for example finite coproducts in much of the author's work with his colleagues Rosebrugh and Dampney). A *specification* is a presentation for a theory, usually given via a *sketch* [1]. A *model* or *state* for a theory is a finite limit preserving, and whatever other exactness properties might have been specified preserving, functor from the theory to the category of finite sets. A model is also called in more mathematical treatments an *algebra*.

A model should be thought of as a snapshot of the system in operation, while the theory constrains the possible snapshots — in a sense the theory embodies all of the information that is required in all possible snapshots.

These formalities allow us to be quite precise about our systems, and to begin to analyse them mathematically. but they also have other advantages some of which we will note here:

- The theory is invariant — there are usually many different presentations of the same system, and we don't wish to deal with artifacts of any particular presentation. For any given system the theories will be equivalent categories no matter which presentation is used.

– The theory includes the constraints, axioms or business rules that are important to capture at specification time, and to enforce at operation time. Indeed, the power of the exactness properties permits, via categorical logic, the specification and enforcement of logically delicate constraints.
– The theory can be constructed so as to include mathematical representations of the data and properties that can be derived from particular states (models) of the system. For example, when the system in question is a database, the theory will include (automatically because of the required finite limits) representations of all of the *queries* that can be applied to the database.

Example 1. Figure 1 is fragment of a theory for a health informatics system [5]. Models of this theory include sets representing for example all of the in-patient operations for which details are stored in the system and all of the hospitals for which details are stored in the system, along with a function between them indicating which operation took place in which hospital. The theory itself includes many more base datatypes together with nodes representing all possible queries of the database, and arrows representing all derived operations among datatypes.

In a little more detail:

– The graph shown is a type diagram. The three monic arrows (those with extra tails to indicate that they should be realised as injective functions) indicate subtypes. The other arrows are functions (operations) which given an instance of their domain type will return an instance of their codomain type. The names on nodes and arrows have no formal significance but do indicate the real world semantics being captured in the specification and will be used from now on in our discussion.
– The commutativity of the two triangles represents a typical real-world constraint: Every in-patient operation conducted at a particular hospital by a particular medical practitioner must take place under a practice agreement (a type of contract) between that hospital and that practitioner. If, instead

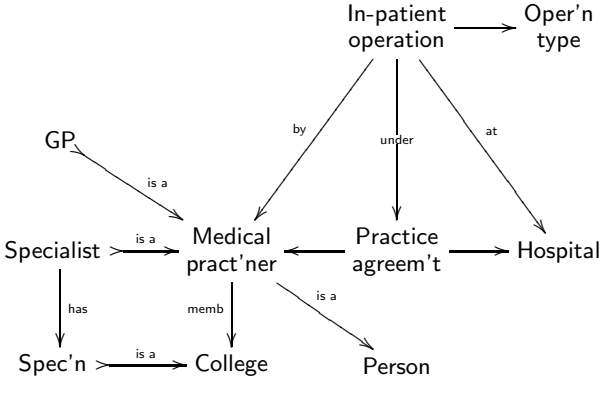


Fig. 1. A fragment of a theory for a health informatics system

the left hand triangle were not required to commute then it would still be the case that every operation took place under an agreement, but Dr A could operate under Dr B's practice agreement. In many information models, situations like this do not even include the arrow marked under, and thus they store the contractual information, but do not specify the constraint — it is expected to be added at implementation time.
- The square is also commutative and is required to be a pullback. This ensures that the specialists are precisely those medical practitioners who are members of a college which occurs in the subtype Specialisation. This is important because the registration procedures (not shown) for specialists are different from those for other medical practitioners.
- Subtype inclusion arrows, and other arrows that are required to be monic in models, are so specified using pullbacks.
- As is common practice, attributes are not shown in Figure 1, but they are important. They are usually large fixed value sets, often of type `integer` (with specified bounding values), `string` (of specified maximum length), `date` etc. Some examples for this theory include the validity period of a practice agreement, the name and the address of a person, the classification of a hospital, the date of an operation, the provider number of a medical practitioner and many more. Strictly, they are all part of the theory.

It is worth noting that in contrast with most algebras that arise in mathematics, information systems are usually very many-sorted (based on many different sets like Hospital, Person, College etc) and most operations are unary (at, by, isa etc). Also in many algebraic specifications essentially unique models (algebras) are sought, frequently by taking initial algebra semantics for example. In contrast ensembles frequently need to collect data maintaining histories or sets of instances for each datatype.

5 Dynamics

The utility of categorical universal algebra for abstract specification of software and systems, including areas as diverse as information systems and programming language semantics, is well-established. So, rather than rehearsing those arguments we will begin by considering one of the limitations of present work — the mathematical treatment of the *dynamics* of algebras is relatively undeveloped in modern universal algebra.

Universal algebra specifies and studies individual algebras, or varieties of algebras with certain properties, but only rarely does it deal with algebras changing through for example adding or deleting an element. Yet this dynamic nature is central for the study of systems that process information — a snapshot of the system at a moment in time is an algebra, and as the system acquires more information, perhaps by the addition of a new instance of some type (eg, an insertion of a new entity instance in a database), the algebra is modified to become another algebra.

One well-understood, but fairly limited, instance of algebra dynamics is the extension of a field or a ring by an indeterminate ($R \longrightarrow R[x]$). In a more general sense algebras can be modified by quotienting or by operating on them with other algebras (taking for example the product — see the next section). But there is little theory to support "Given a group G, add an element with the following properties (expressed in terms of other elements of G) to get a new group G'".

A certain amount of theoretical development of dynamics has been done in [8] which developed a mathematical foundation that unifies the treatment of specifications, updates (dynamics), and categories of algebras for a class of database systems. Nevertheless, much remains to be done.

6 Interactions among System Components

One of the outstanding features of ensembles of computational elements is the interaction of those elements. While ensemble interaction should be dynamic, and possibly adaptive, it is nevertheless important to design and manage interactions and to model them mathematically.

The traditional universal algebra approach to modelling and designing interactions between systems involved calculating pushouts in the category of theories. This has been an effective technique, but it may be seen as less appropriate for ensembles as it views the ensemble as a static system constructed from parts.

R.F.C Walters and his colleagues have an ongoing programme of research into a (bi-) categorical calculus of processes which accurately models the composition of systems including representations of concurrency and feedback (see for example [9] and [10]). The processes may be viewed as algebras in our framework, and algebras can be composed using algebra operations akin to product, sum, and trace. More recently the researchers have incorporated timing issues into their mathematical model. It seems likely that approaches such as these will be very useful in ensemble engineering, although much of the work is still oriented towards viewing the ensemble as a constructed system rather than as a dynamic evolving agglomeration.

Another approach advocated by the author [4] focuses on managing the communications between extant systems using techniques outlined in Section 8. The basis for studying interactions is quite like the pushout approach: We begin with a span of theories $\mathbb{E} \longleftarrow \mathbf{V} \longrightarrow \mathbb{E}'$, two of which, $\mathbb{E}$ and $\mathbb{E}'$, are the theories of independent computational elements while the third ($\mathbf{V}$) is a representation of their interactions, typically the common information on which they will attempt to remain synchronised. The synchronisation techniques (Section 8) are quite different from the calculation of a pushout, but more importantly for this section the systems specified by $\mathbb{E}$ and $\mathbb{E}'$ remain independent and can in principle move in and out of communication rather than being parts of a composite system calculated via pushout.

So, we have at least three promising techniques to analyse and design interactions among elements of ensembles. In all three cases there is still much to do to develop the mathematical approaches to more fully support ensemble engineering.

7 Limited Data at Computational Elements

One of the most significant aspects of an ensemble is the need to deal with semi-structured rather than fully-structured data. In a dynamic world with many adaptive elements joining and leaving an ensemble, it's hard to imagine how complete sets of data can be captured, communicated and maintained.

The problem of incomplete data has long been dealt with in the database community by using NULLs — invalid data values that act as place holders for missing data. Nevertheless, NULLs have had at best an ambiguous status in the theory, frequently they were retro-fitted long after a design which de facto assumed perfect data availability. And when they were considered in the theory they led to widely differing treatments using so-called three-valued logics.

The first observation to be made here is that missing values have no importance in and of themselves — one doesn't need a NULL value to store the fact that there is nothing to store. The delicacy in dealing with missing values arises because *operations* may need to take undefined values.

Now partial operations are easily represented category theoretically. Suppose $f : A \longrightarrow B$ is an operation which might be only partially defined on its domain A. Then in the theory which establishes the type of f we don't include an arrow $A \longrightarrow B$ but rather a span $A \longleftarrow\!\prec A' \longrightarrow B$. Thus A' is the type standing for those instances of A on which f is defined, and notice that in a dynamic system this provides full flexibility — instances of elements of A can be added to, or deleted from, A' as information becomes available.

Interestingly, and delicately, the span approach is not equivalent to using NULLs. In the latter case a partially defined $f : A \longrightarrow B$ would be represented by a fully defined $f' : A \longrightarrow (B + 1)$ where the extra element of the codomain is the NULL. The two approaches are Morita equivalent (ie have equivalent categories of models, see [6]) on the assumption that the subobject $A' \rightarrowtail A$ is complemented and this is not usually the case. The difference is important in dynamic environments: To define f on a new value $a \in A$ is *extra* information and is represented in the span case by inserting a into A', but when f' has codomain $B + 1$ extending the domain of definition really means reassigning values for f' (formerly $f'(a) = 1$, but once f becomes defined at a then $f'(a) = b$ for some $b \in B$).

8 Limited Data during Interoperation

Another source of limited data arises in dynamic environments when, for example, systems interoperate via a span of theories $\mathbb{E} \longleftarrow \mathbf{V} \longrightarrow \mathbb{E}'$ as proposed in Section 6.

Suppose we aim to keep, as far as possible, the algebra for $\mathbb{E}$ synchronised with the algebra for $\mathbb{E}'$ on common parts indicated by $\mathbf{V}$. In categorical universal algebra an algebra for a theory $\mathbb{E}'$ is an appropriate functor $\mathbb{E}' \longrightarrow \mathbf{Set}$. Thus an algebra for $\mathbb{E}'$ yields an algebra for $\mathbf{V}$ by composition with the theory morphism $\mathbf{V} \longrightarrow \mathbb{E}'$. Now, how can we modify the algebra (system snapshot)

for $\mathbb{E}$ so that it remains synchronised with the new algebra for $\mathbf{V}$? This is the *view update problem*. The problem is genuinely difficult because of missing data. A change in the algebra for $\mathbb{E}'$ by an insert say, may result in a change in the algebra for $\mathbf{V}$ also by an insert. That insert is properly defined, since all the information that a $\mathbf{V}$-algebra needs is available in the $\mathbb{E}'$-algebra. But trying to propagate the insert to the extant $\mathbb{E}$-algebra may be impossible (if for example there are constraints that are required to be satisfied by $\mathbb{E}$-algebras that do not appear in $\mathbf{V}$-algebras), or ambiguous (if for example there are operations in $\mathbb{E}$-algebras that are not fully determined by operations in $\mathbf{V}$-algebras).

To engineer effective interoperations we need to determine those occasions when view-updating may be impossible since they are real limitations to interoperations, and for those situations where view-updating may be ambiguous because there are multiple solutions we seek a "best" solution — one for which the change to the $\mathbb{E}$-algebra is minimal. In category theoretic terms we seek a universal solution to the view updating problem and analysing the situation shows that the solution is given by well-known cartesian and op-cartesian morphisms [7].

Importantly the two types of missing data (treated in this section and the preceding section) interact well: In work still being written up the author shows that when operations support missing data using the span approach outlined in the previous section, and an insert leads to an ambiguous view update because such an operation is not fully-determined, the least defined extension of the operation will be a component of an op-cartesian morphism.

Of course there is much work to be done in testing the utility of these approaches for full ensemble engineering, but they have already proved their value in smaller scale system interoperations.

9 Analysing Systems in the Presence of Inconsistencies

Finally we consider one important mathematical limitation that can arise in dealing with ensembles. With loosely coupled, or indeed uncoupled, dynamic systems of open computing elements one can't ensure consistency — unexpected or malfunctioning elements might occasionally join an ensemble and exhibit properties which contradict ensemble invariants. This shouldn't be surprising. Conflicting systems often exist, and frequently operate effectively for extended periods in at least a narrow domain in the real world. But for mathematical tools such inconsistencies can be damning. A single inconsistency in a mathematical model invalidates everything that the model purports to demonstrate.

In fact, it is easy to see how the difference arises. Real systems have various flows of control and inconsistencies can co-exist for extended periods without being invoked together and coming into conflict. The system behaviour is determined by the traces of execution. In contrast mathematical structures exist in their Platonic entirety. If a contradiction exists, its effects cannot be distinguished from deductions that would have been valid in its absence. Every "behaviour" of the mathematical structure, everything that it proves, is brought into doubt by the presence of the inconsistency.

Recent work by Catherine Menon [12] addresses this problem using ideas from view updating. Menon develops what she calls CCF, the categorical consistency framework.

Menon has developed a framework which maintains mathematically the distinctions between modules and permits an analysis of the modules and an exploration of their joint consistency without in fact building them all into a mathematical model which would itself exhibit any inconsistency which was present.

Menon's work is a first step in an area that will need to become well-developed if we are to provide proper mathematical support for truly open and dynamic ensembles rather than using older mathematical techniques to analyse snapshots of ensembles as large static systems constructed from fixed components.

10 Conclusion

It is clear that a great range of mathematical techniques will be important for ensemble engineering. Some exist. Others will be developed to meet the new challenges that arise.

This paper has focused particularly on categorical universal algebra and explored some of the probable applications, and some of the current limitations, of extant universal algebra for ensemble engineering. We've demonstrated that some of the difficult problems of ensemble engineering can be addressed using recent developments based on categorical universal algebra, particularly the representations of missing data and the solution of view update problems, neither of which played a part in earlier applications of universal algebra. We look forward to many more similar developments in mathematical support for ensemble engineering.

References

1. Barr, M., Wells, C.: Category theory for computing science, 2nd edn. Prentice-Hall, Englewood Cliffs (1995)
2. Ehrig, H., Mahr, B.: Fundamentals of algebraic specifications. Springer, Heidelberg (1985)
3. Hölzl, M., Wirsing, M.: State of the Art for the Engineering of Software-Intensive Systems. Deliverable Number D3.1 (2007) (accessed July 19, 2008), http://interlink.ics.forth.gr/central.aspx?sId=84I238I744I323I344283
4. Johnson, M.: Enterprise Software with Half-Duplex Interoperations. In: Doumeingts, G., Mueller, J., Morel, G., Vallespir, B. (eds.) Enterprise Interoperability: New Challenges and Approaches, pp. 521–530. Springer, Heidelberg (2007)
5. Johnson, M., Rosebrugh, R.: View Updatability Based on the Models of a Formal Specification. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, p. 534. Springer, Heidelberg (2001)
6. Johnson, M., Rosebrugh, R.: Three approaches to partiality in the sketch data model. ENTCS 78, 1–18 (2003)

7. Johnson, M., Rosebrugh, R.: Fibrations and Universal View Updatability. Theoretical Computer Science 388, 109–129 (2007)
8. Johnson, M., Rosebrugh, R., Wood, R.J.: Entity-relationship models and sketches. Theory and Applications of Categories 10, 94–112 (2002)
9. Katis, P., Sabadini, N., Walters, R.F.C.: Bicategories of processes. J. Pure Appl. Algebra 115, 141–178 (1997)
10. Katis, P., Sabadini, N., Walters, R.F.C.: On the algebra of systems with feedback and boundary. Rendiconti del Circolo Matematico di Palermo Serie II Suppl. 63, 123–156 (2000)
11. Mac Lane, S.: Categories for the Working Mathematician, 2nd edn. Graduate Texts in Mathematics 5. Springer, Heidelberg (1998)
12. Menon, C.: A category theoretic approach to inconsistencies in modular system specification. PhD thesis, University of Adelaide (2006)
13. Walters, R.F.C.: Categories and Computer Science. Cambridge University Press, Cambridge (1993)

Behaviour Equivalences in Timed Distributed π-Calculus

Gabriel Ciobanu

Romanian Academy, Institute of Computer Science
Blvd. Carol I no.8, 700505 Iaşi, Romania
and "A.I. Cuza" University of Iaşi, Romania
gabriel@iit.tuiasi.ro, gabriel@info.uaic.ro

Abstract. The complexity of the software-intensive systems requires working with notions as explicit locations in a distributed system, interaction among the mobile processes restricted by interaction time-outs, time scheduling, and restricted resource access. In order to work these notions, we use a timed and distributed variant of the π-calculus having explicit locations, types for restricting the resource access, and time constraints for interaction in distributed systems. Using observation predicates, several behavioural notions are defined and related: (global) barbed bisimulations, (global) typed barbed bisimulation, timed (global) barbed bisimulations, timed (global) typed barbed bisimulation and full timed global typed barbed bisimulation. These bisimulations form a lattice according to their distinguishing power.

1 Introduction

Software-intensive systems are complex systems where processes interact with other processes, involving several devices, sensors and applications distributed over heterogeneous networks at various locations. Society's dependence on such software-intensive systems is increasing. Working under dynamic conditions, software-intensive systems should exhibit adaptive and anticipatory behaviour taking care of many aspects involving a spatial distribution and a relative time of interaction among processes. In this paper we explore novel computing models and investigate new behaviour bisimulations for software-intensive systems.

Modelling software-intensive systems requires notions as locations, interaction among the distributed processes, resource access and time scheduling. Starting from the π-calculus [8], distributed π-calculus ($D\pi$) was introduced and studied in [7] as an extension with explicit locations and types. In order to add time constraints for distributed systems, we have introduced timed distributed π-calculus ($tD\pi$) as an extension of the π-calculus with locations, types and timers [3]. The distributed processes are controlled by message communication and time scheduling. Time is important, both for restricting communication availability and for enforcing limited resource access (the communication channels represent resources). $tD\pi$ could be included in the class of control-driven models; the triggering events in $tD\pi$ are either the communications on channels, or the

M. Wirsing et al. (Eds.): Software-Intensive Systems, LNCS 5380, pp. 190–208, 2008.

migration with *go*, or the expiration of a timer. Using timers in coordinating distributed processes, $tD\pi$ goes beyond the coordination by using (only) messages transmitted among processes, or by restricting the actions permitted on channels using channel types. Timers and the time constraints provide temporal synchronisation and scheduling.

Bisimulation is an important concept used for analysing and comparing the processes behaviour; a bisimulation relation equates processes which behave in the same way according to the assumptions defined by the bisimulation. Bisimulations represent the adequate way of capturing behavioural (in)distinguishability of complex systems. The simplicity and richness of the theory of bisimulation made it interesting to define several variants and extensions. The focus of the paper is the behavioural notions and their relations. The paper defines twelve barbed bisimulations, and study their distinguishing power by defining several relationships between these versions of barbed bisimulation for $tD\pi$. The results express the fact that the more aspects are observed, the finer is the corresponding bisimulation. The existence of this family of barbed bisimulations allow to select the right behavioural equivalence depending on the aspects under consideration.

In Section 2 we briefly present the syntax and semantics of the software-intensive model based on the formalism presented in [3]. Some technical results are mentioned: the typing system and typing rules are sound with respect to the dynamic semantics given by reduction rules and equivalence relation. A barb predicate holds if the process respects the observation criteria imposed by its definition. Four barbed bisimulations and eight timed barbed bisimulations are presented in Section 3. The bisimulations are compared by proving first an inclusion with equality, and then giving a counterexample to prove the strict inclusion; the discriminating power of the bisimulations is given by the sets of barbs used by them.

2 A Computation Model for Software-Intensive Systems

The complexity of the software-intensive systems requires working with notions as explicit locations of a distributed system, interaction among the distributed processes restricted by interaction timeouts, time scheduling, and restricted resource access. In order to work these notions, we use a timed and distributed variant of the π-calculus having explicit locations, types for resource access, and time constraints for interaction in distributed systems. Timers over channel names are used in order to define timeouts for communications, and timers over channel types are used in order to restrict their existence inside the type environment of the process. All these timers are decreasing in a uniform way for all the involved clocks (the existence of a global clock is not necessary). The channels are discarded (no communication is possible on these channels) whenever their timers expire; similarly, a channel type is lost when its attached timer expires. In this formalism denoted shortly by $tD\pi$, waiting for a communication on a channel is no longer indefinite (like in π or $D\pi$); if no communication happens

in a predefined interval of time, the waiting process goes to another state. This approach leads to a method of sharing the channels in time.

The timer Δt of each channel makes the channel available for communication only for the period of time determined by the discrete value t. We consider timers for both input and output channels. The reason for adding timers to outputs comes from the fact that in distributed systems we have both multiple clients and multiple servers. This means that clients may switch from one server to another depending on the waiting time. To simplify our presentation we choose a simpler π-calculus and omit the syntax for *matching* or *summation*. A communication channel is considered a fixed resource at a location. The syntax of *Input* and of *Output* communication uses a pair of processes. For instance, an *Input* expression $a^{\Delta t}?(X:T).(P,Q)$ evolves to P whenever a communication is established during the interval of time given by Δt; otherwise it evolves to Q. The variable X is considered bound only in P, and we should provide its type T by defining a typing relation.

Table 1. *Syntax of $tD\pi$*

$u ::= x$	Variable Name	$P, Q ::= stop$	Termination
$\mid a^{\Delta t}$	Timed Channel	$\mid P \mid Q$	Composition
$l ::= x$	Variable Name	$\mid (\nu\, u\colon A)P$	Channel Restriction
$\mid k$	Location Name	$\mid go\, l.(P,Q)$	Movement
$v ::= bv$	Base Value	$\mid u!\langle v\rangle.(P,Q)$	Output
$\mid u \mid l$	Name	$\mid u?(X:T).(P,Q)$	Input
$\mid u@l$	Located Name	$\mid {}^*P$	Replication
$\mid (v_1,..,v_n)$	Tuple of Values	$M, N ::= M \mid N$	Composition
$X ::= x$	Variable	$\mid (\nu\, u@l : T)N$	Located Restriction
$\mid X@l$	Located Variable	$\mid l[[P]]_\Gamma$	Located Process
$\mid (X_1,..,X_n)$	Tuple of Variables		

Two channels are equal $a_1^{\Delta t_1} = a_2^{\Delta t_2}$ if and only if $a_1 = a_2$ and $t_1 = t_2$. Waiting indefinitely on a channel a is allowed by considering Δt as ∞. For example, an output process defined by the expression $a^{\infty}!\langle v\rangle.(P,Q)$ awaits forever to send the value v, simulating the behaviour of an output process in untimed π-calculus. In the expression below, two processes are running in parallel and can interact along the common channel a:

$$a^{\mathbf{\Delta t}}!\langle v\rangle.(P,Q) \mid a^{\mathbf{\Delta t'}}?(X:T).(P',Q') \longrightarrow P \mid P'\{{}^{v}/_{X}\}$$

Each located process is labelled with a type environment Γ which is a set of *location types*. The purpose of the type environment associated with a specific process is to restrict the range of accessible resources the process can access. Formally, $\Gamma \subseteq \mathcal{L} \times K$ is a relation associating to a location name a location type. A location type is a set of *location capabilities* which may contain *channel types*, *move* capability (i.e., permission to migrate to that location), or *channel creation* capability (i.e., permission to create channels). A channel type may contain the channel capabilities *read* (r), *write* (w), and *read only* (ro). A process which has a channel type $res\{r\langle T\rangle, w\langle T'\rangle, ro\langle T''\rangle\}$ can receive messages of type T and send messages of type T'. The ro capability behaves as r with the difference that

the types of the received messages are not added to the type environment of the process. A type environment increases (new types are added) when a name is received along a $r\langle\,\rangle$ channel. With $ro\langle\,\rangle$ capability we describe processes which may use a received channel only if their type environment has a corresponding channel type. In Dπ the resources are accumulated, but they can never be lost (discarded). We extend the channel types of $D\pi$ with timers of the form Δt. Communication is now permitted on channels only in the interval of time given by the timer value t (i.e. until the timer of the channel type expires). These timers define the existence of the channel types inside the type environment. Timers decrease in a uniform way with each "tick" of a clock (this way ensures that we are working uniformly on each location and it is not necessary to consider the existence of a global clock; a synchronization mechanism taking into account two different clocks is enough to get a sound description of the timing aspects). Upon expiration, the channel types are discarded. Timers are created once with the channel types, and are activated when the types are added to the type environment.

The types are presented in [3]. Starting from a set of base types (Integer, Boolean, etc.), a subtyping relation $<:$ is defined similarly to the subtyping relation of $D\pi$ [7]. Note that the intuitive behaviour of the subtyping relation is the inverse of the inclusion of sets ($A <: B$ for types means $A \supset B$ for sets). A process moves to a location (by performing a *go* action), and waits for a period of time to establish a communication on a particular channel; the capabilities $r/w/ro$ for the fixed resources tell a process what is it allowed to do when it reaches a location. When a process receives new channel names, types for the new channels become available. It means that the processes can communicate on the new channels according to the new types. For example, if a process receives through an input channel a located name $a@k$, then it gains the capability to move to location k, and to communicate on channel a. A process which has a channel type with the capability $r\langle T\rangle$ can receive (without generating errors) only messages of type T. When the channel type $res\{r\langle T\rangle\}$ is extended with $r\langle T'\rangle$, it follows naturally that the process is able now to receive messages of a richer type: T and T'. The equality between channel types does not depend on their timers; the equality must be tested only for names and capabilities.

We define a function ψ which affects only the set of capabilities. It decreases the timers of the channel types and removes the types with an expired timer. By removing channel types, it is possible to get location types with only *go* capability (we call them *empty locations*). A process can move to an empty location, but there it does not have the capability to perform any action, and consequently produces a *runtime error*. Thus ψ removes also the empty locations.

Definition 1. *(Cleanup function)*
$\psi : \mathcal{P}_\Delta \to \mathcal{P}_\Delta$ is defined over the set $\mathcal{P}_\Delta$ of located processes such that

$$\psi(l[[P]]_\Gamma) = l[[P]]_{\Gamma'}$$

where l can be any location in the distributed system and Γ' is obtained from Γ where every type $c : res\,\langle\,\rangle\,\Delta t,\ t > 1, t \neq \infty$ is changed to $c : res\,\langle\,\rangle\,\Delta(t-1)$, and every $c : res\,\langle\,\rangle\,\Delta 1$ disappears. Location types of form $loc : \{go\}$ are removed.

Based on the vision that software-intensive systems will contain subsystems using biologically inspired computing principles, we give here an example inspired by the immune systems, showing how to model a complex system, and how a system described in $tD\pi$ is working.

Example 1. The example illustrates the use of the timers introduced in $tD\pi$. T cells are among the most understood cells in the immune system. they are able to destroy the cells to which they bound. Their recognition method uses a T cell receptor (TCR) which is capable to recognize antigens nested in some class of histocompatibility molecules.

A common situation is when an organism gets a virus infection. In this case, the virus invades certain cells in the body. When it gets inside a cell, the virus starts to multiply and, at the end of the process, the whole virus population evades the cell. Early in this process of infection, the cells display fragments of the viral proteins at the surface of the membrane such that T cells can bind to the infected cells and destroy them before the fresh crop of viruses is released. This process can be summarized in the following simplified way: when a virus enters the cell, it offers itself for decoding. If it is read and decoded, then it can be destroyed. But the virus gets stronger in time, and if the decoder does not decode it, after a while the virus stops from offering the reading port and becomes free. A behaviour like the one described above is hard to model with the typical process algebra tools; for $tD\pi$ it illustrates its expressive power.

A molecule infected by a virus offers at a port the possibility to the cytotoxic molecules like *CD8+* to read and initiate the signalling pathway for the destruction sequence before the newly created viruses are released. We call such molecules improperly *antigens* (the $tD\pi$ process is $l_V[[Anti]]_{\Delta_V}$) because, in the first stage of the viral infection, they present at the surface of the membrane binding sites of antigen reminiscences.

$$l_V[[Anti]]_{\Gamma_V}$$

$$Anti = a^{\Delta 9}!\langle inf\rangle.(c^{\Delta 5}?(v:T).(Destroy, Anti), GoFree)$$

$$\Gamma_V = \{l_V : \mathrm{loc}\{a : A, c : A', \ldots\}, \ldots\}$$

$$A = res\langle\rangle\Delta 20$$

The antigen process *Anti* at location l_V sends information on output channel *a!* and awaits for the destroying signal on input channel *c?* for a short period of time (given by the value of its timer $\Delta 5$). The problem is that if the *correct* corresponding T cell does not arrive in time to recognize the virus, this one becomes mature enough and multiply to infect other cells.

Γ_V is the type environment of the located process $l_V[[Anti]]_{\Gamma_V}$; it is basically a set of timed channel permissions which restrict the behaviour of the molecule. For example, for the output channel $a!$ we have two timers: *the channel timer* $\Delta 9$ which restricts the amount of time to wait for an interaction with a T cell, and *the channel type timer* $\Delta 20$ which says that after 20 units of time the virus is mature enough and will no longer offer the output channel for interaction.

We model each molecule type as a process running at a particular location. To simulate the interaction of two molecules, the active one migrates to the location

of the passive one, and interact on a common channel. We define a coordinating process which runs at the special location l_{Co} in parallel with the other processes representing the interacting molecules.

$$Co = C_1 \,|\, C_2 \,|\, \ldots \,|\, C_n$$

Certain initial values are assigned to timers by looking at predetermined known biological facts about the reaction times or the putative times; the reaction with the least putative time is executed.

The T cell *decoders* $d_i[[TDec_i]]_{\Delta_{d_i}}$ decode information and send back answers. We present here the interaction of the T cells with some fictive decoders. Their natural behaviour is more or less the same, as T cells work to recognize viruses. The T cell molecules are

$$l_{A_i}[[CD8_i]]_{\Delta_{A_i}}$$
$$CD8_i = go\, l_V.a^{\Delta 7}?(i:I).go\, d_i.b^{\Delta 5}!\langle i\rangle.d^{\Delta 5}?(c@l : A@K).(go\, l.c^{\infty}!\langle Destroy\rangle, stop)$$
$$\Delta_{A_i} = \{l_V : \mathrm{loc}\ \{a : A\}, d_i : \mathrm{loc}\{\ldots\}, \ldots\}$$

We have several kinds of T cells denoted by $CD8_i$, each of them running at their specific locations l_{A_i}. They go at the location of the antigen l_V and receive on the input channel a? the information about the infecting virus. After receiving the information, a $CD8_i$ migrates to the location d_i of the decoder and retransmits the information for decoding, after which it awaits for an answer which will start or not the destroying sequence of the infected cell. By assigning different timer values to each type of T cell molecule, we get different behaviours for each cell. The $CD8$ process with the lowest value on channel a interacts with the virus. This is a possible strategy in biology, where for the most common viruses we find many compatible T cells. We consider the following system composed of new virus and several T cells and decoders:

$$l_C[[Co]]_{\Delta_C} \,|\, l_V[[Anti]]_{\Delta_V} \,|\, l_{A_1}[[CD8_1]]_{\Delta_{A_1}} \,|\, \ldots$$
$$\ldots \,|\, l_{A_7}[[CD8_7]]_{\Delta_{A_7}} \,|\, \ldots d_9[[TDec_9]]_{\Delta_{d_9}}$$

Let us suppose we have a new virus such that only $TDec_6$ understands its definition. The process $CD8_6$ has a high timer value for channel a, and so it cannot interact with the virus until we reach a situation where the type of channel a is $A = res\langle\rangle\Delta 1$ and the timer on the input channel a? of $CD8_3$ is less than the timer of $CD8_6$. We have an interaction between the antigen and $CD8_3$:

$$l_V[[Anti]]_{\Delta_V} \,|\, l_{A_3}[[CD8_3]]_{\Delta_{A_3}} \,|\, \ldots \quad \longrightarrow$$

Since the timer of the output channel a! of the antigen expires, its channel type A is lost and the time-stepping function ϕ_Δ changes the process $Anti$ to $GoFree$, because no more interactions can be achieved on channel a.

$$l_V[[GoFree]]_{\Delta_V} \,|\, l_{A_6}[[CD8_6]]_{\Delta_{A_6}} \,|\, \ldots$$

2.1 Semantics

The passage of time is formalised by a *time-stepping function* ϕ_Δ defined over the set $\mathcal{P}_\Delta$ of located processes. The possible communications are performed at

every tick of the clock. Active channels are those that could be involved in these communications. ϕ_Δ affects the active channels which do not communicate at the tick of the clock (the channels involved in communication disappear together with their timers). Due to timers, the capabilities can be lost, which leads to "errors". We define ϕ_Δ to check the existence of the needed types and change the process accordingly. As ϕ_Δ decreases the channel timers we also extend it to take care of the type environments by applying the cleanup function ψ. In the definition of ϕ_Δ we omit the channel type and the transmitted message in the input and output processes for brevity.

Well-typedness of processes is defined by a set of static rules; a detailed presentation of these rules is given in [3]. These rules express the behaviour of a process with regard to its types. If a process wants to communicate on a channel for which it has no capability, it can still be well-typed if the so-called *safety process* Q is well-typed.

Definition 2. *(Time-stepping function)*
We define $\phi_\Delta : \mathcal{P}_\Delta \to \mathcal{P}_\Delta$, *where* Γ' *is obtained by application of the cleanup function* ψ. *Note that we use a concise notation* $a^{\Delta t}.(R,Q)$ *to stand for both* $a^{\Delta t}!\langle v\rangle.(R,Q)$ *and* $a^{\Delta t}?(X:T).(R,Q)$.

$$\phi_\Delta(l[[P]]_\Gamma) = \begin{cases} k[[R]]_{\Gamma'} & \text{if } P \equiv go\,k.(R,Q) \text{ and } \Gamma(k) <: loc\{go\} \\ l[[Q]]_{\Gamma'} & \text{if } P \equiv go\,k.(R,Q) \text{ and } k \notin dom(\Gamma) \\ l[[a^{\Delta(t-1)}.(R,Q)]]_{\Gamma'} & \text{if } P \equiv a^{\Delta t}.(R,Q),\ t>1 \text{ and } t \neq \infty \\ l[[Q]]_{\Gamma'} & \text{if } P \equiv a^{\Delta t}.(R,Q),\ t \leq 1 \\ l[[Q]]_{\Gamma'} & \text{if } P \equiv a^{\Delta t}.(R,Q),\ t>1 \text{ and } \Gamma \not<: \Gamma(l,a) \\ \phi_\Delta(l[[R]]_\Gamma) \mid \phi_\Delta(l[[Q]]_\Gamma) & \text{if } P \equiv R \mid Q \\ (\nu a@l:A)\phi_\Delta(l[[R]]_{\Gamma\{a@l:A\}}) & \text{if } P \equiv (\nu a:A)R \\ l[[P]]_{\Gamma'} & \text{otherwise} \end{cases}$$

We write $\Gamma \vdash P$ and say that *process P is well-typed with respect to type environment Γ*; we also write $\Gamma \vdash_k P$ and say that *P is well-typed to run at location k*. To say that $P \equiv a^{\Delta t}!\langle v\rangle.(R,Q)$ is well-typed to run at location k with respect to type environment Γ, the following statements should hold: *(i)* $\Gamma \vdash_k v{:}T$ which means that v is a well-formed value at location k of type T; *(ii)* $\Gamma \vdash_k a : res\{w\langle T\rangle\}\Delta t$ which means that channel a exists at location k and may communicate values of type T for another t units of time; *(iii)* $\Gamma \vdash_k R$; $\Gamma \vdash_k Q$ which means that R and Q are well-typed at location k.

Since function ψ changes the capability set Γ by removing channel and location types, we are interested if the process is still well-typed under the new Γ'. The following lemma relates the typing environment of the processes with the passage of time. For complete proofs see [3].

Lemma 1. *(Well-typedness is preserved by the cleanup function)*
If $\Gamma \Vdash l[[P]]_\Delta$ *then* $\Gamma \Vdash \psi(l[[P]]_\Delta)$.

We consider the located processes ranged over by N and M (e.g., N represents $l[[P]]_\Gamma$). We denote by $\not\to$ the fact that rules (R_Γ-COM1) and (R_Γ-COM2) cannot be applied. Using these notations, the operational semantics of $tD\pi$ is given by the smallest relation defined by the following reduction rules:

$$(\mathrm{R}_\Gamma\text{-IDLE})\ \frac{l[[P]]_\Gamma \not\rightarrow}{l[[P]]_\Gamma \rightarrow \phi_\Delta(l[[P]]_\Gamma)}$$

$$(\mathrm{R}_\Gamma\text{-COM1})\ \frac{\Gamma(l,a) <: res\{r\langle T\rangle\}}{\begin{array}{c} l[[a^{\Delta t}!\langle v\rangle.(P,Q)]]_\Delta \mid l[[a^{\Delta t'}?(X:T).(P',Q')]]_\Gamma \rightarrow \\ \psi(l[[P]]_\Delta) \mid \psi(l[[P'\{^v/_X\}]]_{\Gamma \sqcap \{v@l:T\}}) \end{array}}$$

$$(\mathrm{R}_\Gamma\text{-COM2})\ \frac{\Gamma(l,a) <: res\{ro\langle T\rangle\}}{\begin{array}{c} l[[a^{\Delta t}!\langle v\rangle.(P,Q)]]_\Delta \mid l[[a^{\Delta t'}?(X:T).(P',Q')]]_\Gamma \rightarrow \\ \psi(l[[P]]_\Delta) \mid \psi(l[[P'\{^v/_X\}]]_\Gamma) \end{array}}$$

$$(\mathrm{R}_\Gamma\text{-PAR})\ \frac{N \rightarrow N' \quad M \rightarrow M'}{N \mid M \rightarrow N' \mid M'} \qquad (\mathrm{R}_\Gamma\text{-RES})\ \frac{N \rightarrow N'}{(\nu a@l:T)N \rightarrow (\nu a@l:T)N'}$$

$$(\mathrm{R}_\Gamma\text{-CONG})\ \frac{N \equiv N' \quad N \rightarrow M \quad M \equiv M'}{N' \rightarrow M'}$$

In (R_Γ-IDLE) the function ϕ_Δ decreases the timers on channels, and for the expired timers the function discards the channels. We have two communication rules which depend on the type of the channel. In (R_Γ-COM2) we consider $ro\langle\,\rangle$ channels, and the process may use the received information without adding the new type to its type environment Γ, as the case in rule (R_Γ-COM1). In these cases the type environments are affected by the cleanup function ψ. In rule (R_Γ-PAR) a process M reduces to M' by (R_Γ-IDLE) rule if it has no internal communication reductions. Because the movement syntax enters under the application of function ϕ_Δ, we have no (R_Γ-GO) rule. At each tick of the clock (R_Γ-IDLE) is applied to *go* processes and to processes which do not enter any communication.

A run-time error system of $tD\pi$ is presented in [3] as a set of rules describing a relation $\xrightarrow{err}$ denoting the generation of an error. A run-time error occurs only when the channel or location type is in the type environment (when a process tries to do something against the types accumulated in its type environment).

The soundness of the new formalism is given by the following results. We follow a method introduced in [5]; the proofs can be found in [3].

Lemma 2. *If* $\Gamma \Vdash l[[P]]_\Delta$ *then* $\Gamma \Vdash \phi_\Delta(l[[P]]_\Delta)$.

Theorem 1. *(Subject Reduction) For all located processes*

(a) If $N \equiv N'$, *then* $\Gamma \Vdash N$ *if and only if* $\Gamma \Vdash N'$.
(b) If $\Gamma \Vdash N$ *and* $N \rightarrow N'$, *then* $\Gamma \Vdash N'$.

Subject reduction ensures that once well-typed, a process remains well-typed. Contrary to the general approach in functional programming, in $tD\pi$ well-typedness must be preserved also by the structural equivalence relation. In the following we give a result of *type safety* which is needed to get a complete proof of the soundness property of $tD\pi$. This result states that if a system is well-typed, then it cannot give rise to run-time errors, and this is denoted by $P \overset{err}{\not\longrightarrow}$.

Theorem 2. *For all located processes* N *and all type environments* Γ *such that* $\Gamma \Vdash N$ *we have* $N \overset{err}{\not\longrightarrow}$.

3 Bisimulations in Software-Intensive Systems

Whenever the operational semantics is defined by a reduction relation (i.e., no labels over transitions), the observation predicates are fundamental ingredients in describing the behaviour. They allow to observe the interaction capabilities of the processes, and to compare the evolution of two systems. There are mainly four observation coordinates in $tD\pi$: one involves the name of the communication channel (Milner and Sangiorgi's barbed bisimulation), another is given by the locations, a third one is given by the type environment, and finally, a forth one is given by time. Several barbed bisimulations can be defined. Two processes are barbed bisimilar if they satisfy the same observation predicates and, by performing a reduction, can evolve to processes that are still barbed bisimilar. The main purpose of this section is to emphasize the diversity of choices offered by $tD\pi$ for comparing the behaviour of processes. We define four barbed bisimulations which can be found in the literature in similar forms, and add several new extensions which involve time. Finally, a total of twelve barbed bisimulations form a lattice presented in Figure 1.

Following the presentation of barbed bisimulation in [9], we specify first what is observable, and what is unobservable. To simplify the presentation we choose as observable only the communication along the located channel names, without considering the transmitted messages. In $tD\pi$ we have synchronous communication on fixed located channels. In consequence, the observables can be both *input* and *output* communications. We consider as unobservable the *movement* with *go*, the *application of the time-stepping function* ϕ_Δ, and the *internal interaction* of processes. For instance, in order to observe if a process P communicates on the input channel name $a?$ at location k, the observer waits for an output communication along the same channel name $a!$ at the same location k. We consider that an *untimed observer* cannot distinguish the values of the timers on channel names or on channel types. On the other hand, a *timed observer* can trace the timers of the channel types inside the type environment, and/or the timers of the channel names.

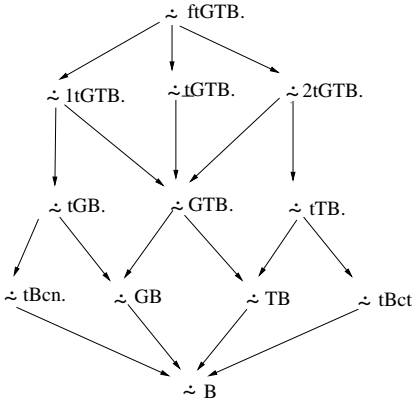


Fig. 1. Barbed bisimulations for $tD\pi$ and their related distinguishing power

Definition 3. *A barb predicate* $\downarrow_\mu$ *where* $\mu \in \{a?, a!\}$ *with* a *being any channel name, is defined inductively by the following system of rules where we denote by* $\underline{\mu}$ *the names of the input or output channels (e.g. if* $\mu = a?$ *then* $\underline{\mu} = a$*).*

$$\frac{}{a^{\Delta t}!\langle v\rangle.(P,Q)\downarrow_{a!}} \quad \frac{}{a^{\Delta t}?(X:T).(P,Q)\downarrow_{a?}}$$

$$\frac{P\downarrow_\mu}{P\,|\,Q\downarrow_\mu} \quad \frac{P\downarrow_\mu \text{ and } a \neq \underline{\mu}}{(\nu\, a:A)P\downarrow_\mu} \quad \frac{P\downarrow_\mu}{{}^*P\downarrow_\mu}$$

Additionally, a process P *satisfies the predicate* $\Downarrow_\mu$*, denoted* $P\Downarrow_\mu$*, if and only if there is a sequence of reductions* $P \twoheadrightarrow Q$ *leading from* P *to* Q*, and* $Q\downarrow_\mu$*.*

Definition 4. *A barbed bisimulation* $\mathcal{S}$ *is a symmetric binary relation over processes that for all* $P, Q \in \mathcal{S}$ *implies*

if $P\downarrow_\mu$*, then* $Q\downarrow_\mu$ *for any barb* $\downarrow_\mu$*;*

if $P \to P'$*, then* $Q \to Q'$ *and* $(P', Q') \in \mathcal{S}$*. Two processes are barbed bisimilar, denoted* $P \dot{\sim}_B Q$*, if and only if* $(P, Q) \in \mathcal{S}$ *for some barbed bisimulation* $\mathcal{S}$*.*

The barbs and the barbed bisimulation are naturally applied to located processes N. The barbed bisimulation $\dot{\sim}_B$ is the coarsest bisimulation in this paper; the observer is restricted to observe only communications. A general notion of barbs is given in [6] in terms of *acceptances* $s_1 A_1 \ldots s_n A_n$, where s_i is a sequence of actions from the set of actions Act and $A_i \subset Act$. These general barbs correspond in LTS semantics to the *failure traces* of van Glabbeek [11]. However we want to look at actions separately, not at sequences s_i of actions. In consequence we adopt a presentation based on the notion of barbs introduced in [9]. Note that, even if in $tD\pi$ we do not use the summation operator, a process P may satisfy more than one barb because of the parallel composition operator. Thus, if $P\downarrow_\mu$ and $Q\downarrow_{\mu'}$ with $\mu \neq \mu'$, then both statements $P\,|\,Q\downarrow_\mu$ and $P\,|\,Q\downarrow_{\mu'}$ hold.

The weak version of the barbed bisimulation for $tD\pi$ makes less strict the constraints of the strong bisimulation. The extension to weak barbed bisimulation is general, and it can be applied to any barbed bisimulation by replacing the barb $Q\downarrow_\mu$ with its weak variant $Q\Downarrow_a$, and the one-step reduction relation $Q \to Q'$ with its closure $Q \twoheadrightarrow Q'$. We denote the weak barbed bisimulation by $\dot{\approx}_B$.

The barbed bisimulation by itself does not offer satisfactory properties. In order to obtain a barbed equivalence (barbed congruence), the bisimulation is closed under all static (respectively normal) contexts [10]. A context could be viewed as a process running in parallel with the equated processes. In [10] it is shown that in the setting of the π-calculus, the barbed equivalence and barbed congruence coincide with labelled early bisimilarity, respectively congruence relation on the class of image-finite processes [11].

We focus on other possible barbed bisimilarities. Having locations and fixed located communication channels, it is rather natural to strengthen the observing power of the previous defined barbs with locations.

Definition 5. *A global barb predicate* $\downarrow_{\mu@k}$ *with* μ *representing input or output communication (as described in Definition 3) is defined inductively by the following rules where* N *and* M *are processes located at location* k*, and* l *is any location of the system, including* k*.*

$$\overline{k[[a^{\Delta t}!\langle v\rangle.(P,Q)]]\downarrow_{a!@k}} \quad \overline{k[[a^{\Delta t}?(X:T).(P,Q)]]\downarrow_{a?@k}}$$

$$\frac{N\downarrow_{\mu@k}}{N\mid M\downarrow_{\mu@k}} \quad \frac{N\downarrow_{\mu@k} \text{ and } a\neq\mu}{(\nu\, a@l:A)N\downarrow_{\mu@k}} \quad \frac{k[[P]]\downarrow_{\mu@k}}{k[[^*P]]\downarrow_{\mu@k}}$$

The *global barbed bisimulation* $\dot{\sim}_{GB}$ and its weak variant $\dot{\approx}_{GB}$ are defined in the same way as the simple barbed bisimulations given above. A global barbed bisimulation compares processes by looking also at the location of the communication. This is of interest when we want to trace the trip of the processes through the distributed system. Two global barbed bisimilar processes migrate through the same set of locations in the same order.

We use $\mathcal{O}$ for the semantics induced by the barbed bisimulation $\dot{\sim}_{\mathcal{O}}$, and $\mathcal{O}^{\approx}$ for the weak case $\dot{\approx}_{\mathcal{O}}$. We give a preorder relation $\prec$ over barbed bisimulations to represent that the first barbed bisimulation makes at least as much identifications as the second. We write $\mathcal{O}\prec\mathcal{O}'$ instead of $\dot{\sim}_{\mathcal{O}}\prec\dot{\sim}_{\mathcal{O}'}$. By definitions, it is clear that $\mathcal{O}^{\approx}\prec\mathcal{O}$. Note that in the definition of the barbed bisimulations we always use the same reduction relation of $tD\pi$. The difference in observational power between the barbed bisimulations comes from the definition of the observation predicates (barbs).

Proposition 1. *($B\prec GB$) The global barbed bisimulation is strictly finer than the barbed bisimulation:*

1. *$B\preceq GB$, i.e. $\forall N,M$, if $N\dot{\sim}_{GB}M$ then $N\dot{\sim}_{B}M$*
2. *$GB\not\preceq B$, i.e. $\exists N,M$, s.t. if $N\dot{\sim}_{B}M$ then $N\not\dot{\sim}_{GB}M$*

Proof: The proofs for this kind of statements are similar; first an inclusion with equality, and then a counterexample to prove the strict inclusion. Bisimulation relations are represented and studied as sets of process pairs. Therefore the comparisons between bisimilarities are based on set-theoretic comparisons. For the first part of the proof, the global observer (i.e., the global barb) can distinguish in both processes the same communication on channel a at the same location k. Therefore the normal observer has the same barbs on channel a in both processes, and so the global barb implies the normal barb ($N\downarrow_{\mu@k}\Rightarrow N\downarrow_{\mu}$).

Counterexample 1: Take two located processes N and M:

$$N=l[[a^{\Delta t}!\langle v\rangle.(P,Q)]]_{\Gamma} \quad \text{and} \quad M=k[[a^{\Delta t}!\langle v'\rangle.(P,Q)]]_{\Gamma'}$$

Both $N\downarrow_{a!}$ and $M\downarrow_{a!}$ hold, and thus the two processes are barbed bisimilar: $N\dot{\sim}_{B}M$. $N\downarrow_{a!@l}$ and $M\downarrow_{a!@k}$ hold; however $l\neq k$, and so $N\not\dot{\sim}_{GB}M$. □

Another feature of $tD\pi$ is the type system used to restrict the access to resources. In the definition of *typed barbs* the observability of types is restricted by the observers distinguishing power over types. A typed barb $\downarrow^{\Gamma}_{\mu:A}$ identifies a process which can communicate on channel with the name μ and has enough permissions determined by the channel type A with respect to the type environment Γ of the observer.

Definition 6. *A typed barb predicate $\downarrow^{\Gamma}_{\mu:A}$ is defined inductively by the following rules where N and M are located processes (possibly located at different locations).*

$$\frac{\Gamma'(k,a) <: \Gamma(k,a) \quad \Gamma(k,a) = A}{k[[a^{\Delta t}!\langle v\rangle.(P,Q)]]_{\Gamma'} \downarrow^{\Gamma}_{a!:A}} \qquad \frac{\Gamma'(k,a) <: \Gamma(k,a) \quad \Gamma(k,a) = A}{k[[a^{\Delta t}?(X:T).(P,Q)]]_{\Gamma'} \downarrow^{\Gamma}_{a?:A}}$$

$$\frac{N \downarrow^{\Gamma}_{\mu:A}}{N \mid M \downarrow^{\Gamma}_{\mu:A}} \qquad \frac{N \downarrow^{\Gamma}_{\mu:A} \text{ and } a \neq \underline{\mu}}{(\nu\, a@l : A)N \downarrow^{\Gamma}_{\mu:A}} \qquad \frac{k[[P]]_{\Gamma'} \downarrow^{\Gamma}_{\mu:A}}{k[[{}^*P]]_{\Gamma'} \downarrow^{\Gamma}_{\mu:A}}$$

The definition of the *typed barbed bisimulation* $\dot{\sim}_{TB}$ follows Definition 4, replacing the barb $\downarrow_{\mu}$ with the typed barb $\downarrow^{\Gamma}_{\mu:A}$. To obtain the desired equivalence we should close the barbed bisimulation under all contexts. Equivalently, we can close the barbed bisimulation under all observers well-typed with respect to the type environment Γ. Thus N and M are *typed barbed equivalent* (and we write $N \sim_{TB} M$) iff we have $N \mid O \dot{\sim}_{TB} M \mid O$ for all O with $\Gamma \Vdash O$.

Proposition 2. *($B \prec TB$) The typed barbed bisimulation is strictly finer than the barbed bisimulation:*

1. *$B \preceq TB$, i.e. $\forall N, M$, if $N \dot{\sim}_{TB} M$ then $N \dot{\sim}_{B} M$*
2. *$TB \not\preceq B$, i.e. $\exists N, M$, s.t. if $N \dot{\sim}_{B} M$ then $N \not\dot{\sim}_{TB} M$*

The proofs of the following results are quite similar to the line described in the proof of Proposition 1, and they are omitted.

The global barbed bisimulation $\dot{\sim}_{GB}$ and the typed barbed bisimulation $\dot{\sim}_{TB}$ are incomparable. Based on the distinction between locations and types, we can build two processes which are equated by $\dot{\sim}_{GB}$, but not by $\dot{\sim}_{TB}$, and another example for the opposite direction. A finer bisimulation is obtain by combining locations and types into *global typed barbed bisimulation* $\dot{\sim}_{GTB}$. The *global typed barbs* are essentially typed barbs which can also observe the location of the communication channel. In order to choose a clear notation for global typed barbs $\downarrow^{\Gamma}_{\mu@k}$, we direct the reader to the Definition 6 of the typed barbs. We have the type A deduced from the fact that the global typed barb is restricted to the type environment Γ, and the communication channel $\underline{\mu}$ is local to k; hence $A = \Gamma$.

Definition 7. *A global typed barb predicate $\downarrow^{\Gamma}_{\mu@k}$ is defined inductively by the following rules; N and M are located processes at the same location k.*

$$\frac{\Gamma'(k,a) <: \Gamma(k,a)}{k[[a^{\Delta t}!\langle v\rangle.(P,Q)]]_{\Gamma'} \downarrow^{\Gamma}_{a!@k}} \qquad \frac{\Gamma'(k,a) <: \Gamma(k,a)}{k[[a^{\Delta t}?(X:T).(P,Q)]]_{\Gamma'} \downarrow^{\Gamma}_{a?@k}}$$

$$\frac{N \downarrow^{\Gamma}_{\mu@k}}{N \mid M \downarrow^{\Gamma}_{\mu@k}} \qquad \frac{N \downarrow^{\Gamma}_{\mu@k} \text{ and } a \neq \underline{\mu}}{(\nu\, a@l : A)N \downarrow^{\Gamma}_{\mu@k}} \qquad \frac{k[[P]]_{\Gamma'} \downarrow^{\Gamma}_{\mu@k}}{k[[{}^*P]]_{\Gamma'} \downarrow^{\Gamma}_{\mu@k}}$$

The definition of the *global typed barbed bisimulation* $\dot{\sim}_{GTB}$ follows Definition 4, by replacing the barb $\downarrow_\mu$ with the global typed barb $\downarrow^{\Gamma}_{\mu@k}$.

Proposition 3. *(GB $\prec$ GTB) The global typed barbed bisimulation is strictly finer than the global barbed bisimulation:*

1. *GB $\preceq$ GTB, i.e. $\forall N, M$, if $N \dot{\sim}_{GTB} M$ then $N \dot{\sim}_{GB} M$*
2. *GTB $\not\preceq$ GB, i.e. $\exists N, M$, such that if $N \dot{\sim}_{GB} M$, then $N \dot{\not\sim}_{GTB} M$*

Proposition 4. *(TB $\prec$ GTB) The global typed barbed bisimulation is strictly finer than the typed barbed bisimulation:*

1. *TB $\preceq$ GTB, i.e. $\forall N, M$, if $N \dot{\sim}_{GTB} M$ then $N \dot{\sim}_{TB} M$*
2. *GTB $\not\preceq$ TB, i.e. $\exists N, M$, such that if $N \dot{\sim}_{TB} M$ then $N \dot{\not\sim}_{GTB} M$*

It is clear by definition that both $N \downarrow^{\Gamma}_{a!:A}$ and $M \downarrow^{\Gamma}_{a!:A}$ hold, and thus $N \dot{\sim}_{TB} M$. However the locations of the communication channels are different ($l \neq k$), and thus $N \downarrow^{\Gamma}_{a!@k}$ holds, but $M \downarrow^{\Gamma}_{a!@k}$ does not hold, i.e., $N \dot{\not\sim}_{GTB} M$.

The timed features of the new computational model are emphasized by considering timed observers able to check the values of the timers.

Definition 8. *A timed barb predicate $\downarrow^{t_{cn}}_{\mu}$ distinguishes the communication channel and its timer value, and is defined inductively by the following rules ($Q \not\downarrow^{t'_{cn}}_{\mu}$ means that the timed barb predicate $\downarrow^{t'_{cn}}_{\mu}$ does not hold for process Q).*

$$\frac{}{a^{\Delta t}!\langle v\rangle.(P,Q)\downarrow^{t_{cn}}_{a!}} \qquad \frac{}{a^{\Delta t}?(X:T).(P,Q)\downarrow^{t_{cn}}_{a?}}$$

$$\frac{P\downarrow^{t_{cn}}_{\mu} \quad Q \not\downarrow^{t'_{cn}}_{\mu}\ \forall t' \in \mathbb{N}}{P \mid Q \downarrow^{t_{cn}}_{\mu}} \qquad \frac{P\downarrow^{t_{cn}}_{\mu} \quad Q\downarrow^{t'_{cn}}_{\mu} \quad t_{cn} < t'_{cn}}{P \mid Q \downarrow^{t_{cn}}_{\mu}}$$

$$\frac{P\downarrow^{t_{cn}}_{\mu} \text{ and } a \neq \mu}{(\nu\, a:A)P\downarrow^{t_{cn}}_{\mu}} \qquad \frac{P\downarrow^{t_{cn}}_{\mu}}{{}^{*}P\downarrow^{t_{cn}}_{\mu}}$$

When treating the parallel composition operator, we choose the smallest timer value. If Q can offer the communication channel a for a shorter time, then after the expiration of the timer $\Delta t'$, Q would change state by application of ϕ_{Δ}. The new state Q' may offer a complementary communication channel for P, and thus the system composed of P and Q' can no longer offer a communication on channel a. We say that our timed barbs offer *consistent observations*.

Definition 9. *A timed barbed bisimulation for channel names $\mathcal{S}$ is a symmetric binary relation over processes which for all $(P,Q) \in \mathcal{S}$ implies*

1. *if $P\downarrow^{t_{cn}}_{\mu}$, then $Q\downarrow^{t_{cn}}_{\mu}$ for any timed barb $\downarrow^{t_{cn}}_{\mu}$;*
2. *if $P \rightarrow P'$ then $Q \rightarrow Q'$, and $(P', Q') \in \mathcal{S}$.*

Two processes are timed barbed bisimilar, denoted $P \dot{\sim}_{tBcn} Q$, if and only if $(P,Q) \in \mathcal{S}$ for some timed barbed bisimulation $\mathcal{S}$.

Proposition 5. *(B $\prec$ tBcn) The timed barbed bisimulation for channel names is strictly finer than the barbed bisimulation:*

1. $B \preceq tBcn$, *i.e.* $\forall P, Q$, *if* $P \dot{\sim}_{tBcn} Q$ *then* $P \dot{\sim}_B Q$
2. $tBcn \not\preceq B$, *i.e.* $\exists P, Q$, *such that if* $P \dot{\sim}_B Q$ *then* $P \dot{\not\sim}_{tBcn} Q$

We extend the timed barb with location awareness, obtaining a finer *timed global barb* denoted by $\downarrow^t_{\mu@k}$ which identifies the channel μ and its timer value t, and also the location k of the channel. This barb induces a *timed global barbed bisimulation* $\dot{\sim}_{tGB}$.

Definition 10. *A timed global barb predicate* $\downarrow^t_{\mu@k}$ *is defined inductively by the following rules, where* N *and* M *are located processes at location* k*, and* l *can be any location.*

$$\frac{}{k[[a^{\Delta t}!\langle v\rangle.(P,Q)]]\downarrow^t_{a!@k}} \qquad \frac{}{k[[a^{\Delta t}?(X:T).(P,Q)]]\downarrow^t_{a?@k}}$$

$$\frac{N\downarrow^t_{\mu@k} \quad M\not\downarrow^{t'}_{\mu@k}\ \forall t' \in \mathbb{N}}{N \mid M\downarrow^t_{\mu@k}} \qquad \frac{N\downarrow^t_{\mu@k} \quad M\downarrow^{t'}_{\mu@k}\ t < t'}{N \mid M\downarrow^t_{\mu@k}}$$

$$\frac{N\downarrow^t_{\mu@k} \text{ and } a \neq \mu}{(\nu\, a@l : A)N\downarrow^t_{\mu@k}} \qquad \frac{k[[P]]\downarrow^t_{\mu@k}}{k[[^*P]]\downarrow^t_{\mu@k}}$$

Proposition 6. *(tBcn* $\prec$ *tGB) The timed global barbed bisimulation is strictly finer than the timed barbed bisimulation:*

1. $tBcn \preceq tGB$, *i.e.* $\forall N, M$, *if* $N \dot{\sim}_{tGB} M$ *then* $N \dot{\sim}_{tBcn} M$
2. $tGB \not\preceq tBcn$, *i.e.* $\exists N, M$, *such that if* $N \dot{\sim}_{tBcn} M$, *then* $N \dot{\not\sim}_{tGB} M$

Proposition 7. *(GB* $\prec$ *tGB) The timed global barbed bisimulation is strictly finer than the global barbed bisimulation:*

1. $GB \preceq tGB$, *i.e.* $\forall N, M$, *if* $N \dot{\sim}_{tGB} M$ *then* $N \dot{\sim}_{GB} M$
2. $tGB \not\preceq GB$, *i.e.* $\exists N, M$, *s.t. if* $N \dot{\sim}_{GB} M$ *then* $N \dot{\not\sim}_{tGB} M$

Since we also have timers on channel types, we give a second timed variant $\downarrow^{t_{ct}}_{\mu}$ of the simple barbs which takes into account only the type timers. The rules for this timed barb are similar to the ones in Definition 8. Following Definition 9, we define *timed barbed bisimulation for channel types* denoted $\dot{\sim}_{tBct}$ induced by these barbs.

Proposition 8. *(B* $\prec$ *tBct) The timed barbed bisimulation for channel types is strictly finer than the normal barbed bisimulation:*

1. $B \preceq tBct$, *i.e.* $\forall N, M$, *if* $N \dot{\sim}_{tBct} M$ *then* $N \dot{\sim}_B M$
2. $tBct \not\preceq B$, *i.e.* $\exists N, M$, *s.t. if* $N \dot{\sim}_B M$ *then* $N \dot{\not\sim}_{tBct} M$

Timed extensions of the typed barbs could be given by looking at the channel type timer. Our *timed typed barbed bisimulation* below cannot be compared with $\dot{\sim}_{tBcn}$ because the two bisimulations look at different timers.

Definition 11. *A timed typed barb predicate* $\downarrow^{t\Gamma}_{\mu:A}$*, where* $\mu \in \{a!, a?\}$ *and* $\Gamma(k, a) = A$*, is defined inductively by the following rules.*

$$\frac{\Gamma'(k,a) <: \Gamma(k,a) \quad \Gamma(k,a) = res\{\ldots\}\Delta t}{k[[a^{\Delta s}!\langle v\rangle.(P,Q)]]_{\Gamma'} \downarrow^{t\Gamma}_{a!:A}} \qquad \frac{\Gamma'(k,a) <: \Gamma(k,a) \quad \Gamma(k,a) = res\{\ldots\}\Delta t}{k[[a^{\Delta s}?(X:T).(P,Q)]]_{\Gamma'} \downarrow^{t\Gamma}_{a?:A}}$$

$$\frac{N \downarrow^{t\Gamma}_{\mu:A} \quad M \not\downarrow^{t'\Gamma}_{\mu:A} \quad \forall t' \in \mathbb{N}}{N \mid M \downarrow^{t\Gamma}_{\mu:A}} \qquad \frac{N \downarrow^{t\Gamma}_{\mu:A} \quad M \downarrow^{t'\Gamma}_{\mu:A} \quad t < t'}{N \mid M \downarrow^{t\Gamma}_{\mu:A}}$$

$$\frac{N \downarrow^{t\Gamma}_{\mu:A} \text{ and } a \neq \mu}{(\nu\, a@l : A)N \downarrow^{t\Gamma}_{\mu:A}} \qquad \frac{k[[P]]_{\Gamma'} \downarrow^{t\Gamma}_{\mu:A}}{k[[{}^*P]]_{\Gamma'} \downarrow^{t\Gamma}_{\mu:A}}$$

We have used s as a value for the channel timer. However we do not take into account the channel timer Δs. The definition of the *timed typed barbed bisimulation* $\dot{\sim}_{tTB}$ follows Definition 4 by replacing $\downarrow_\mu$ with the timed typed barb $\downarrow^{t\Gamma}_{\mu:A}$.

Proposition 9. *(tBct $\prec$ tTB) The timed typed barbed bisimulation is strictly finer than the timed barbed bisimulation for channel types:*

1. *tBct $\preceq$ tTB, i.e. $\forall P, Q$, if $P \dot{\sim}_{tTB} Q$ then $P \dot{\sim}_{tBct} Q$*
2. *tTB $\not\preceq$ tBct, i.e. $\exists P, Q$, s.t. if $P \dot{\sim}_{tBct} Q$ then $P \dot{\not\sim}_{tTB} Q$*

Proposition 10. *(TB $\prec$ tTB) The timed typed barbed bisimulation is strictly finer than the typed barbed bisimulation:*

1. *TB $\preceq$ tTB, i.e. $\forall P, Q$, if $P \dot{\sim}_{tTB} Q$ then $P \dot{\sim}_{TB} Q$*
2. *tTB $\not\preceq$ TB, i.e. $\exists P, Q$, s.t. if $P \dot{\sim}_{TB} Q$ then $P \dot{\not\sim}_{tTB} Q$*

To compare processes by considering all the features of the formalism, we should observe the communication channel name and its location, the type of the channel, and also the values of the timers. As we have seen, depending on which timers we are interested in observing, we obtain different types of timed barbed bisimulations, and these bisimulations are incomparable.

We have two different timed global typed barb predicates: $\downarrow^{\Gamma}_{t,\mu@k}$ which distinguishes only the channel timer, and $\downarrow^{t\Gamma}_{\mu@k}$ which distinguishes only the type timer. These barbs induce two *partial timed global typed barbed bisimulations* denoted by $\dot{\sim}_{1tGTB}$ and $\dot{\sim}_{2tGTB}$, respectively.

Proposition 11. *(tGB $\prec$ 1tGTB) The timed global typed barbed bisimulation of rank I is strictly finer than the timed global barbed bisimulation:*

1. *tGB$\preceq$1tGTB, i.e. $\forall P, Q$, if $P \dot{\sim}_{1tGTB} Q$ then $P \dot{\sim}_{tGB} Q$*
2. *1tGTB $\not\preceq$ tGB, i.e. $\exists P, Q$, s.t. if $P \dot{\sim}_{tGB} Q$ then $P \dot{\not\sim}_{1tGTB} Q$*

Proposition 12. *(GTB $\prec$ 1tGTB) The timed global typed barbed bisimulation of rank I is strictly finer than the global typed barbed bisimulation:*

1. *GTB $\preceq$ 1tGTB, i.e. $\forall P, Q$, if $P \dot{\sim}_{1tGTB} Q$ then $P \dot{\sim}_{GTB} Q$*
2. *1tGTB $\not\preceq$ GTB, i.e. $\exists P, Q$, such that if $P \dot{\sim}_{GTB} Q$ then $P \dot{\not\sim}_{1tGTB} Q$*

Proposition 13. *(tTB $\prec$ 2tGTB) The timed global typed barbed bisimulation of rank II is strictly finer than the timed typed barbed bisimulation:*

1. *tTB$\preceq$ 2tGTB, i.e. $\forall P, Q$, if $P \dot{\sim}_{2tGTB} Q$ then $P \dot{\sim}_{tTB} Q$*
2. *2tGTB $\not\preceq$ tTB, i.e. $\exists P, Q$, such that if $P \dot{\sim}_{tTB} Q$ then $P \dot{\not\sim}_{2tGTB} Q$*

A *timed global typed barbed bisimulation* is denoted by $P \dot{\sim}_{\underline{t}LTB} Q$; it takes into account both the timer on channel name and on channel type, but chooses only the one with the smallest value. It is defined as in Definition 4; here we give only the definition of the associated barb.

Definition 12. *A timed global typed barb predicate, $\downarrow^{t\Gamma}_{\mu@k}$ is defined inductively by the following system of rules. We remind that we denote by $\underline{\mu}$ the name of the output or input channels $\{a!, a?\}$; where by $\underline{t}$ we denote only a discrete value of the timer; $\underline{t} = min(t, t')$.*

$$\frac{\Gamma'(k,a) = res\{\ldots\}\Delta t' \quad \Gamma'(k,a) <: \Gamma(k,a)}{k[[a^{\Delta t}!\langle v\rangle.(P,R)]]_{\Gamma'} \downarrow^{\underline{t}\Gamma}_{a!@k}} \qquad \frac{\Gamma'(k,a) = res\{\ldots\}\Delta t' \quad \Gamma'(k,a) <: \Gamma(k,a)}{k[[a^{\Delta t}?(X:T).(P,R)]]_{\Gamma'} \downarrow^{\underline{t}\Gamma}_{a?@k}}$$

$$\frac{N\downarrow^{t\Gamma}_{\mu@k} \quad M\not\downarrow^{t'\Gamma}_{\mu@k} \ \forall t' \in \mathbb{N}}{N \mid M \downarrow^{t\Gamma}_{\mu@k}} \qquad \frac{N\downarrow^{t\Gamma}_{\mu@k} \quad M\downarrow^{t'\Gamma}_{\mu@k} \ t < t'}{N \mid M \downarrow^{t\Gamma}_{\mu@k}}$$

$$\frac{N\downarrow^{\underline{t}\Gamma}_{\mu@k} \ and \ a \neq \underline{\mu}}{(\nu\, a@l : A)N \downarrow^{\underline{t}\Gamma}_{\mu@k}} \qquad \frac{k[[P]]\downarrow^{\underline{t}\Gamma}_{\mu@k}}{k[[^*P]]\downarrow^{\underline{t}\Gamma}_{\mu@k}}$$

We decide to use the minimum between the two timer values because of the application of the time-stepping function ϕ_Δ which changes the process P to R after the expiration of the smallest timer. Looking at a channel a with a channel timer Δt and type timer $\Delta t'$, we can distinguish the following two cases:

i) $t > t'$: the type is removed from the type environment after t' units of time, and we have the following case of Definition 2 of ϕ_Δ: $l[[P]]_\Gamma = l[[a^{\Delta t}.(Q,R)]]_\Gamma$, $t > 1$, and $\Gamma \not<: \Gamma(l,a)$;
ii) $t' > t$: the timer of the channel a expires after t units of time, and we have the following case of Definition 2 of ϕ_Δ: $l[[P]]_\Gamma = l[[a^{\Delta t}.(Q,R)]]_\Gamma$ and $t \leq 1$.

In both cases $\phi_\Delta(l[[P]]_\Gamma) = l[[R]]_{\Gamma'}$.

Proposition 14. *(GTB $\prec$ $\underline{t}$GTB) The timed global typed barbed bisimulation is strictly finer than the global typed barbed bisimulation:*

1. *GTB$\preceq$ $\underline{t}$GTB, i.e. $\forall P, Q$, if $P \dot{\sim}_{\underline{t}GTB} Q$ then $P \dot{\sim}_{GTB} Q$*
2. *$\underline{t}$GTB$\not\preceq$GTB, i.e. $\exists P, Q$, such that if $P \dot{\sim}_{GTB} Q$ then $P \dot{\not\sim}_{\underline{t}GTB} Q$*

1tGTB, 2tGTB, and $\underline{t}$GTB are incomparable (counterexamples can show that processes which can be equated by one barbed bisimulation cannot be equated by the other).

The *full timed global typed barbed bisimulation* has the greatest discriminating power among the bisimulations presented in this paper; it defines the largest number of equivalence classes over processes.

Definition 13. *A full timed global typed barb predicate,* $\downarrow_{\mu@k}^{t,t'\Gamma}$ *is defined inductively by the following system of rules:*

$$\frac{\Gamma'(k,a) <: \Gamma(k,a) \quad \Gamma(k,a) = res\{\ldots\}\Delta t'}{k[[a^{\Delta t}!\langle v\rangle.(P,R)]]_{\Gamma'} \downarrow_{a!@k}^{t,t'\Gamma}} \qquad \frac{\Gamma'(k,a) <: \Gamma(k,a) \quad \Gamma(k,a) = res\{\ldots\}\Delta t'}{k[[a^{\Delta t}?(X:T).(P,R)]]_{\Gamma'} \downarrow_{a?@k}^{t,t'\Gamma}}$$

$$\frac{N \downarrow_{\mu@k}^{t,t'\Gamma} \quad M \downarrow_{\mu@k}^{t,t'\Gamma}}{N \mid M \downarrow_{\mu@k}^{t,t'\Gamma}} \qquad \frac{N \downarrow_{\mu@k}^{t,t'\Gamma} \quad and \quad a \neq \underline{\mu}}{(\nu\, a@l : A)N \downarrow_{\mu@k}^{t,t'\Gamma}} \qquad \frac{k[[P]] \downarrow_{\mu@k}^{t,t'\Gamma}}{k[[*P]] \downarrow_{\mu@k}^{t,t'\Gamma}}$$

The definition of the full timed global typed barbed bisimulation $\dot{\sim}_{ftGTB}$ follows Definition 4, replacing the barb $\downarrow_\mu$ by $\downarrow_{\mu@k}^{t,t'\Gamma}$. As the full barb traces both the values of the timers on the channel name and channel type, it is simple to prove that 1tGTB $\prec$ ftGTB, 2tGTB $\prec$ ftGTB and $\underline{t}$GTB $\prec$ ftGTB.

Proposition 15. *($\underline{t}GTB \prec ftGTB$) The full timed global typed barbed bisimulation is strictly finer than the timed global typed barbed bisimulation:*

1. $\underline{t}GTB \preceq ftGTB$, *i.e.* $\forall P, Q$, *if* $P \dot{\sim}_{ftGTB} Q$ *then* $P \dot{\sim}_{\underline{t}GTB} Q$
2. $ftGTB \not\preceq \underline{t}GTB$, *i.e.* $\exists P, Q$, *such that if* $P \dot{\sim}_{\underline{t}GTB} Q$ *then* $P \dot{\not\sim}_{ftGTB} Q$

Proof: For the first part of the proof we observe that the barb $\downarrow_{\mu@k}^{t,t'\Gamma}$ distinguishes the values of both timers. If $P \dot{\sim}_{ftGTB} Q$ implies that $P \downarrow_{\mu@k}^{t,t'\Gamma}$ and $Q \downarrow_{\mu@k}^{t,t'\Gamma}$ hold. This means that the values of the channel type timers are equal, and also the values of the channel name timers are equal; this means that the two minimum values are also equal. This implies that both processes respect the same timed global typed barb $\downarrow_{\mu@k}^{\underline{t}\Gamma}$; thus $N \downarrow_{\mu@k}^{t,t'\Gamma} \Rightarrow \downarrow_{\mu@k}^{\underline{t}\Gamma}$. For the second part we can give a counterexample where the located processes are equated by the barbed bisimulation $\dot{\sim}_{\underline{t}GTB}$. The same processes do not have corresponding equal timer values, and therefore they are not equated by $\dot{\sim}_{ftGTB}$. □

4 Conclusion

Timed distributed π-calculus is presented as a computation model of software-intensive systems. It has explicit notions of location and time, it deals explicitly with resources allocation, secure resource usage and management. Important constraints in new distributed applications are given by timeout coordination realized by a discrete and relative notion of time given by various types of timers. A non-monotonic behaviour of the systems are given by the timeout recovery processes. We use an example to describe the process coordination in time and space, suggesting how to use this model to model an adaptive behaviour in open distributed systems.

We introduce timed distributed π-calculus in [3] as a timed and distributed extension of the π-calculus. The essential ingredients are represented by timers

on channels and channel types. Thus $tD\pi$ combines the temporal constraints with types and locations in order to give the possibility of modelling located and timed interactions between distributed processes with time restricted resource access. In this paper, after presenting briefly the syntax and semantics of the timed distributed π-calculus, we define several observation predicates and their corresponding barbed bisimulations, establishing relations between them. We emphasise the diversity of choices for several behavioural equivalences which can be defined in the framework of $tD\pi$ calculus. The defined bisimulations involve locations, types, timers on channels, and timers on channel types. The bisimulations are organised in a lattice (Figure 1) according to their distinguishing power. Depending on the aspect we intend to observe, a suitable barbed bisimulation can be selected from this lattice. The lattice can be used to reason about the distinguishing power of such bisimulations, comparing them and finding the right place in the lattice of any bisimulation defined for distributed systems with locations, types and timers.

The coordination aspects of the timed distributed π-calculus are discussed in [2] where a student is moving from one location to another location by interacting with a bus, a tram, and two cabs. The system is described both in timed distributed π-calculus and timed mobile ambients. Timed mobile ambients are presented in [1], and they are essentially mobile ambients with types and timers. The main difference between timed distributed π-calculus and timed mobile ambients is given by the representation of space ; the space used in $tD\pi$ is flat, while in timed mobile ambients we have a more realistic account of physical distribution by using a hierarchical representation of space.

Acknowledgements

Many thanks to my former student Cristian Prisacariu for his contribution during the initial steps of this work. A short preliminary version was presented and published at ICCP 2006 [4].

References

1. Aman, B., Ciobanu, G.: Mobile Ambients with Timers and Types. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) ICTAC 2007. LNCS, vol. 4711, pp. 50–63. Springer, Heidelberg (2007)
2. Ciobanu, G.: Interaction in Time and Space. Electronic Notes in Theoretical Computer Science 203, 5–18 (2008)
3. Ciobanu, G., Prisacariu, C.: Timers for Distributed Systems. Electronic Notes in Theoretical Computer Science 164, 81–99 (2006)
4. Ciobanu, G., Prisacariu, C.: Barbed Bisimulations for Timed Distributed π-Calculus. In: 2nd Int'l Conference on Intelligent Computer Communication and Processing, Cluj-Napoca, pages 6 (2006)
5. Felleisen, M., Wright, A.K.: A Syntactic Approach to Type Soundness. Information and Computation 115, 38–94 (1994)

6. Hennessy, M., Regan, T.: A Process Algebra For Timed Systems. Information and Computation 117, 221–239 (1995)
7. Hennessy, M., Riely, J.: Resource Access Control in Systems of Mobile Agents. Information and Computation 173, 82–120 (2002)
8. Milner, R.: Communicating and Mobile Systems: the π-calculus. Cambridge University Press, Cambridge (1999)
9. Milner, R., Sangiorgi, D.: Barbed Bisimulation. LNCS, vol. 623, pp. 685–695. Springer, Heidelberg (1992)
10. Sangiorgi, D.: Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms, PhD Thesis, University of Edinburgh (1992)
11. van Glabbeek, R.J.: The Linear Time - Branching Time Spectrum. Handbook of Process Algebra, pp. 3–99 (2001)

The Chemical Reaction Model Recent Developments and Prospects

Jean-Pierre Banâtre[1], Pascal Fradet[2], and Yann Radenac[1,⋆]

[1] Université de Rennes 1 and INRIA
IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
{jbanatre,yradenac}@irisa.fr
[2] INRIA
Inria Grenoble, 655 avenue de l'Europe, 38330 Montbonnot, France
Pascal.Fradet@inria.fr

Abstract. In 2001, we gave a survey of more than fifteen years of research on the chemical paradigm which had been a source of inspiration in many different research areas. The present article presents a digest of recent advances concerning the chemical reaction model. We focus to a large extent on: (1) upgrading the basic model to a higher order formalism allowing reactions to be part of solutions and to take part in reactions and (2) generalizing standard multisets to hybrid and infinite multisets, thus providing new forms of interactions between elements. These novelties, incorporated in the HOCL language (High Order Chemical Language), provide natural and elegant ways of expressing properties related to coordination and self-organization of systems. Finally, we present current research directions which strive to make the chemical reaction model effective particularly in the programming of large-scale, highly parallel applications such as Grids.

1 Introduction

Recent years showed that new software intensive systems, such as service-oriented architectures [1], web services and cloud computing [2], are entering the main stream. Such distributed systems spread over the Internet and have more and more users, resources and services. As their popularity grows and the size of their applications scales up, their programming becomes a major challenge. Sequential languages do not fit whereas parallel languages based on explicit coordinated processes are too complex when considering a huge number of processes. Still, most current approaches are based on conventional programming languages extended by a library that let the programmer tackle low level issues such as the distribution over the resources or the management of the dynamicity and failures of these systems.

⋆ Current address: Research Center for Grid and Service Computing, Institute of Computing Technology, Academy of Sciences, Beijing 100080, P.R. China. yann.radenac@software.ict.ac.cn. This author was partially supported by the EchoGRID IST project n°045520, funded by the European Commission.

M. Wirsing et al. (Eds.): Software-Intensive Systems, LNCS 5380, pp. 209–234, 2008.

Another approach is investigated here: instead of extending standard programming languages, we develop and use a high level unconventional programming language. Recent work on "chemical" programming [3,4,5] showed that it could easily describe grid and autonomous systems. In this article, we review the chemical model of computation, present recent developments, case studies and research directions which show that chemical programming is an interesting candidate to specify software intensive systems.

The Gamma formalism was proposed in [6] as a new paradigm for parallel computing. Basically, it captures the intuition of computation as the global evolution of a collection of atomic values interacting freely. Gamma can be introduced intuitively through the chemical reaction metaphor. The unique data structure in Gamma is the multiset which can be seen as a chemical solution. A simple program is made of a reaction condition and an action. Execution proceeds by replacing elements satisfying the reaction condition by the elements specified by the action. The result of a Gamma program is obtained when a stable state is reached, that is to say, when no more reactions can take place.

For example, the computation of the maximum element of a non empty multiset can be described by the reaction rule:

$$\textbf{replace}\, x, y \,\textbf{by}\, x \,\textbf{if}\, x \geq y$$

meaning that any couple of elements x and y of the multiset is replaced by x if the condition is fulfilled. This process goes on till a stable state is reached, that is to say, when only the maximum element remains. Note that, in this definition, nothing is said about the order of evaluation of the comparisons. If several disjoint pairs of elements satisfy the condition, the reactions can be performed in parallel.

The literature about Gamma has been quite prolific and, for a large part, is summarized in [7]. In the last five years, a strong momentum has led to several extensions of the basic concept of multiset on two respects: the nature of these elements and the multiplicity of its elements. The first generalization consists of allowing elements of multisets to be chemical programs (or reactions) themselves. On a computational viewpoint, this leads to a higher order chemical language, thus allowing explicit manipulation of chemical programs though chemical reactions. This extension is formalized by the γ-calculus, a minimal higher-order calculus that summarizes in a few rules the essence of higher-order chemical programming. By extending that calculus with constants, operators, types and expressive patterns, a higher-order chemical programming language (HOCL) can be built.

The second extension generalizes the multiplicity of multiset elements. We consider hybrid multisets [8] which can contain elements with either positive or negative multiplicity. An element x with a negative multiplicity $-k$ is seen as an anti-element able to annihilate k occurrences of x. We consider also elements with an infinite multiplicity called multiplets. For example, the multiplet 1^{∞} represents a multiset containing an unbounded number of 1's. This can be used to represent an element which can be part of an arbitrary number of simultaneous parallel reactions.

Section 2 summarizes the state of affairs as of year 2001 (a Gamma story). Section 3 defines the fundamentals of chemical programming with the presentation of the γ-calculus, the λ-calculus of chemical programming. We proceed by extending that simple model into an expressive Higher-Order Chemical Language (here starts the HOCL story), which is enriched with generalized multisets. Section 4 show how HOCL can express classical synchronization schemes and autonomous systems. Section 5 focuses on more prospective research concerned with the design of effective chemical programming systems. Section 6 gives our personal insight for the future of this approach and concludes.

2 The Chemical Reaction Model

Initially, the chemical reaction model was proposed by Jean-Pierre Banâtre and Daniel Le Métayer in [9] through the Gamma programming model. Thereafter, Gamma has inspired many contributions [7].

2.1 Gamma

The Gamma [6] formalism was proposed to capture the intuition of computation as the global evolution of a collection of atomic values interacting freely. Gamma is a kernel language which can be introduced intuitively through the chemical reaction metaphor. The unique data structure in Gamma is the multiset which can be seen as a chemical solution. A simple program is a pair (Reaction condition, Action). Execution proceeds by replacing in the multiset elements satisfying the reaction condition by the products of the action. The result is obtained when a stable state is reached, that is to say when no more reactions can take place. The following is an example of a Gamma program computing the maximum element of a non-empty set.

$$max = \textbf{replace}\, x,\ y\, \textbf{by}\, y\, \textbf{if}\, x \leq y$$

The side condition $x \leq y$ specifies a property to be satisfied by the selected elements x and y. These elements are replaced in the set by the value y. Nothing is said in this definition about the order of evaluation of the comparisons. If several disjoint pairs of elements satisfy the condition, the reactions can be performed in parallel. In order to write a program computing the maximum of a set of values in a "traditional" language, we would first have to choose a representation for the set. This representation could typically be an array for an imperative language or a list for a declarative language. The program would be defined as an iteration through the array, or a recursive walk through the list. The important point is that the data structure would impose constraints on the order in which elements are accessed. Of course, parallel versions of imperative or functional programs can be defined (solutions based on the "divide and conquer" paradigm for example), but none of them can really model the total absence of ordering between elements that is achieved by the Gamma program. The essential feature of the Gamma programming style is that a data structure is no longer seen as

a hierarchy that has to be walked through or decomposed by the program in order to extract atomic values. Atomic values are gathered into one single bag and the computation is the result of their individual interactions. A related notion is the "locality principle" in Gamma: individual values may react together and produce new values in a completely independent way. As a consequence, a reaction condition cannot include any global condition on the multiset such as $\forall$-properties or properties on the cardinality of the multiset. The locality principle is crucial because it makes it easier to reason about programs and it encapsulates the intuition that there is no hidden control constraints in Gamma programs.

Let us now consider the problem of computing the prime numbers less than a given value n. The basic idea of the algorithm can be described as follows: "start with the set of values from 2 to n and remove from this set any element which is the multiple of another element". So the Gamma program is built as the sequential composition of *iota* which computes the set of values from 2 to n and *rem* which removes multiples. The program *iota* itself is made of two reactions: the first one splits an interval $x{:}y$ with $x = y$ in two parts and the second one replaces any interval $x{:}x$ by the value x.

$$
\begin{aligned}
primes(n) &= rem(iota(\{2{:}n\})) \\
iota &= \mathbf{replace}\, x{:}y \,\mathbf{by}\, x{:}((x+y)/2),\ (((x+y)/2)+1){:}y \,\mathbf{if}\, x \neq y \\
&\quad\ \mathbf{replace}\, x{:}y \,\mathbf{by}\, x \,\mathbf{if}\, x = y \\
rem &= \mathbf{replace}\, x,\ y \,\mathbf{by}\, y \,\mathbf{if}\, multiple(x, y)
\end{aligned}
$$

The first reaction increases the size of the multiset, the second one keeps it constant and the third one makes the multiset shrink. In contrast with the usual sequential or parallel solutions to this problem (usually based on the successive application of sieves [6]), the Gamma program proceeds through a collection of atomic actions applying on individual and independent pieces of data. The program does not introduce any constraint on the way the comparisons are carried out.

A small number of program schemes are indeed necessary to write most applications. Five schemes (basic reactions), called *tropes* (for Transmuter, Reducer, OPtimiser, Expander, Selector) are particularly useful. For example, a transmuter is a rule that transform one molecule into another one, a reducer is a rule that outputs less molecules than the number of molecules that have reacted, etc. Further details about tropes may be found in [10].

The interested reader can find in [6] a longer series of examples chosen from a wider range of domains: string processing problems, graph problems, geometric problems. Gamma has also been used in a project aiming at experimenting high-level programming languages for prototyping image processing applications [11,12]. In [13], an operating system kernel is defined in Gamma and proven correct.

2.2 Implementations

A property of Gamma which is often presented as an advantage is its potential for concurrent interpretation. In principle, due to the locality property, each

tuple of elements fulfilling the reaction condition can be handled simultaneously. It should be clear however that managing all this parallelism efficiently can be a difficult task and complex choices have to be made in order to map the chemical model on parallel architectures. Several parallel implementations have been proposed:

- *Distributed memory implementations.* Two protocols have been proposed [14,15] for the distributed implementation of Gamma on network of communicating machines. They differ in the way rewritings are controlled. either centralized or distributed [16,17].
- *Shared memory implementations.* The multiset is the unique data structure from which elements are extracted and where elements resulting from the reaction are stored. A specific software architecture has been developed in [18] in order to provide an efficient Gamma implementation on a Sequent multiprocessor machine.
- *Hardware implementation.* The tropes defined above have been used as a basis for the design of a specialized hardware architecture [19]. A hardware skeleton is associated with each trope and these skeletons are parametrized and combined according to the program to be implemented.

2.3 Linguistic Extensions

Several linguistic extensions of Gamma have been proposed: composition operators for Gamma, higher-order Gamma, and Structured Gamma.

For the sake of modularity, it is desirable that a language offer a rich set of operators for combining programs. [20] presents of a set of operators for Gamma and studies their semantics and the corresponding calculus of programs. The two basic operators considered in this paper are the sequential composition $P_1 \circ P_2$ and the parallel composition $P_1 + P_2$. The intuition behind $P_1 \circ P_2$ is that the stable multiset reached after the execution of P_2 is given as argument to P_1. On the other hand, the result of $P_1 + P_2$ is obtained (roughly speaking) by executing the reactions of P_1 and P_2 (in any order, possibly in parallel), terminating only when neither can proceed further.

Another approach for the introduction of composition operators in a language consists in providing a way for the programmer to define them as higher-order programs. This is the traditional view in the functional programming area and it requires to be able to manipulate programs as ordinary data. This is the approach followed in [21] which proposes a higher-order version of Gamma. Contrary to HOCL (see Section 3), reactions are kept separate from the multiset and cannot be considered as first-class citizens.

The choice of the multiset as the unique data constructor is central in the design of Gamma. However, this may lead to programs which are unnecessary complex when the programmer needs to encode specific data structures. For example, it is necessary to resort to pairs $(index, value)$ to represent sequences. The solution proposed in [22] is based on a notion of structured multiset which can be seen as a set of addresses satisfying specific relations and associated with

a value. Types, defined as graph grammars, characterize precisely the structure of the multiset. Structured Gamma allows the programmer to define his own types and have his programs checked according to the type definitions.

2.4 Gamma as a Bridge between Specifications and Implementations

Until now, we have presented Gamma as a programming language. Gamma can also be seen as a very high-level language bridging the gap between specification languages and low-level (implementation oriented) languages.

In order to prove the correctness of a program in an imperative language, a common practice consists in splitting the property into two parts: the invariant which holds during the whole computation, and the variant which is required to hold only at the end of the computation. In the case of total correctness, it is also necessary to prove that the program must terminate. The important observation concerning the variant property is that a Gamma program terminates when no more reaction can take place, which means that no tuples of elements satisfy the reaction condition. So we obtain the variant of the program by taking the negation of the reaction condition. In order to prove the termination of the program, we have to provide a well-founded ordering (an ordering such that there is no infinite descending sequences of elements) and to show that the application of an action decreases the multiset according to this ordering. To this aim, we can resort to a result from [23] allowing the derivation of a well-founded ordering on multisets from a well-founded ordering on elements of the multiset.

A method for the derivation of Gamma programs from specifications in first order logic is proposed in [24]. The basic strategy consists in splitting the specification into a conjunction of two properties which will play the roles of the invariant and the variant of the program to be derived. The interested reader can find a more complete treatment of several examples in [24].

As mentioned earlier, the philosophy of Gamma is to introduce a clear separation between correctness issues and efficiency issues in program design. In particular, Gamma can be seen as a specification language which does not introduce unnecessary sequentiality. As a consequence, designing a reasonably efficient implementation of the language is not straightforward. The hardest problem concerns the construction of all tuples to be checked for reaction. A blind approach to this problem leads to an intractable complexity but a thorough analysis of the possible relationships between the elements of the multiset and the shape of the reaction condition may lead to improvements which highly optimize the execution and produce acceptable performances. In his thesis, C. Creveuil [25] studied several optimizations to refine Gamma programs into well-known versions of sequential algorithms. Several proposals have been made to enrich Gamma with features which could be exploited by a compiler to reduce the overhead associated with the "magic stirring" process. For example, the language of *schedules* [26] provides extra information about control in Gamma programs, and *local linear logic* [27] as well as Structured Gamma [22] structure the multiset. We come back to these issues when we present our future prospects (Section 5).

2.5 Gamma as of Source of Inspiration

The chemical reaction model has served as the basis of a number of works in various, often unexpected, research directions.

The Chemical Abstract Machine. The chemical abstract machine (or Cham) [28] was proposed by Berry and Boudol to describe the operational semantics of process calculi. The most important additions to Gamma are the notions of membrane and airlock mechanism. Membranes are used to encapsulate solutions and to force reactions to occur locally. In terms of multisets, a membrane can be used to introduce multiset of molecules inside a multiset that is to say "to transform a solution into a single molecule" [29]. The airlock mechanism is used to describe communications between an encapsulated solution and its environment. The Cham was used in [28] to define the semantics of various process calculi (TCCS, Milners π-calculus of mobile processes) and a concurrent lambda calculus. A Cham for the call-by-need reduction strategy of λ-calculus is defined in [29]. The Cham has inspired a number of other contributions.

Shape Types. The work around Structured Gamma showed that many data structures could be described as graph grammars and manipulated by reactions. Shape-C is an extension of C which integrates the notion of types as graph grammars (called here shapes) and reactions. The notion of graph grammars is powerful enough to describe most complex data structures (see [30] for a description of skip lists, red-black trees, left-child-right-sibling trees in terms of graph grammars). Due to their precise characterization of data structures, shape types are a very useful facility for the construction of safe programs.

Software Architectures. Another related area of application which has attracted a great amount of interest is the formal definition of software architectures. Typical examples of software architectures are the "client-server organization", "layered systems", "blackboard architecture". Despite the popularity of this topic, little attention has focused on methods for comparing software architectures or proving that they satisfy certain properties. The chemical reaction model has been used for specifying software architectures [31] and architecture styles [32]. One major benefit of the approach is that it makes it possible to define several architectures for a given application and compare them in a formal way.

Influences of the chemical reaction model can be found in other domains such as visual languages [33], protocols for shared virtual memories [34] or logic programming [35].

3 Higher-Order Chemical Model

In the higher-order chemical model, reaction rules are considered as molecules inside the chemical solution. This feature has lead to the computation model called the γ-calculus, which has been extended as the HOCL programming language, and which has been extended again by generalizing multiplicities.

$$
\begin{array}{lll}
M ::= & x & \text{; variable} \\
\mid & \gamma\langle x\rangle.M & \text{; } \gamma\text{-abstraction} \\
\mid & M_1, M_2 & \text{; multiset} \\
\mid & \langle M\rangle & \text{; solution}
\end{array}
$$

$$
\begin{array}{lcll}
(\gamma\langle x\rangle.M), \langle N\rangle & \longrightarrow & M[x := N] \quad \text{if } \mathit{Inert}(N) & \text{; } \gamma\text{-reduction} \\
M_1, M_2 & \equiv & M_2, M_1 & \text{; commutativity} \\
M_1, (M_2, M_3) & \equiv & (M_1, M_2), M_3 & \text{; associativity}
\end{array}
$$

Fig. 1. Syntax and rules of γ-terms (i.e., molecules)

3.1 The γ-Calculus

The γ-*calculus* [36] can be seen as a formal and minimal basis for the chemical paradigm in much the same way as the λ-calculus is the formal basis of the functional paradigm.

The fundamental data structure of the γ-calculus is the multiset. Computation can be seen either intuitively, as chemical reactions of elements agitated by Brownian motion, or formally, as higher-order, associative and commutative (AC), multiset rewritings. The syntax of γ-terms (also called *molecules*) is given in Fig. 1. A γ-abstraction is a reactive molecule which consumes a molecule (its argument) and produces a new one (its body). Molecules are composed using the AC multiset constructor ",". A solution encapsulates molecules and keeps them separate. It serves to control and isolate reactions.

The γ-calculus bears clear similarities with the λ-calculus. They both rely on the notions of (free and bound) variable, abstraction and application. A λ-abstraction and a γ-abstraction both specify a higher-order rewrite rule. However, λ-terms are tree-like whereas the AC nature of the application operator "," makes γ-terms multiset-like. Associativity and commutativity (AC) formalize Brownian motion and make the notion of solution necessary, if only to distinguish between a function and its argument.

The conversion rules and the reduction rule of the γ-calculus are gathered in Fig. 1. Chemical reactions are represented by a single rewrite rule, the γ-reduction, which applies a γ-abstraction to a solution. A molecule $(\gamma\langle x\rangle.M), \langle N\rangle$ can be reduced only if the content N of the solution argument is a closed term made exclusively of γ-abstractions or exclusively of solutions (which may be active). So, a molecule can be extracted from its enclosing solution only when it has reached an inert state. This is an important restriction that permits the ordering of rewritings. Without this restriction, the contents of a solution could be extracted in any state and the solution construct would lose its purpose. Reactions can occur in parallel as long as they apply to disjoint sub-terms. A molecule is in normal form if all its molecules are inert.

The λ-calculus can easily be encoded within the γ-calculus (see [36] for more details). In fact, the γ-calculus is more expressive than the λ-calculus since it

$$\begin{array}{ccc} (\gamma\langle x\rangle.\gamma\langle y\rangle.x), \langle A\rangle, \langle B\rangle & \equiv & (\gamma\langle x\rangle.\gamma\langle y\rangle.x), \langle B\rangle, \langle A\rangle \\ \downarrow & & \downarrow \\ (\gamma\langle y\rangle.A), \langle B\rangle & & (\gamma\langle y\rangle.B), \langle A\rangle \\ \downarrow & & \downarrow \\ A & \not\equiv & B \end{array}$$

Fig. 2. The γ-calculus is not confluent

can also express non-deterministic programs. For example, if A and B are two distinct normal forms, then Fig. 2 shows that the γ-calculus is not confluent.

3.2 HOCL : A Higher-Order Chemical Language

The γ-calculus is a quite expressive higher-order calculus. However, it is too low level to be used as a practical programming language (it looks like assembler). Compared to the original Gamma [6] and other chemical models [21,37], it lacks two fundamental features:

- *Reaction condition.* In Gamma, reactions are guarded by a condition that must be fulfilled in order to apply them. Compared to γ where inertia and termination are described syntactically, conditional reactions give these notions a semantic nature.
- *Atomic capture.* In Gamma, any fixed number of elements can take part in a reaction. Compared to a γ-abstraction which reacts with one element at a time, a n-ary reaction takes atomically n elements which cannot take part in any other reaction at the same time.

These two extensions are orthogonal and enhance greatly the expressivity of chemical calculi.

HOCL (*Higher Order Chemical Language*) is a programming language based on the γ-calculus extended with reaction condition, atomic capture, rich pattern language, expressions, types, pairs, empty solutions, naming and a syntax with keywords.

One-shot rules are denoted by $\mathbf{one}\, P\, \mathbf{by}\, M\, \mathbf{if}\, C$ which react with molecules that match the pattern P and satisfy the reaction condition C. They are replaced by the molecule M. Formally, we have:

$$N,\ \mathbf{one}\, P\, \mathbf{by}\, M\, \mathbf{if}\, C \longrightarrow \phi M \qquad \text{if } \phi = match(P, N) \wedge \phi C$$

where ϕ is the substitution obtained by matching P with N.

Many programs are naturally expressed by applying the same reaction an arbitrary number of times. HOCL introduces n-shot rules which are not consumed by the reaction (they correspond to fix point of abstractions in the γ-calculus). We denote them by the same syntax as in Gamma:

$$\mathbf{replace}\, P\, \mathbf{by}\, M\, \mathbf{if}\, C$$

Such a molecule reacts like one-shot rules but it remains in the solution after the reaction and can be used as many times as necessary.

$$N,\ \textbf{replace}\ P\ \textbf{by}\ M\ \textbf{if}\ C \longrightarrow \phi M, \textbf{replace}\ P\ \textbf{by}\ M\ \textbf{if}\ C \qquad \text{if } \phi = P \text{ match } N \wedge \phi C$$

A HOCL program is an unstable solution of molecules. The execution of that program consists in performing the reactions (modulo AC) until a stable state is reached (i.e., no more reaction can occur). A standard Gamma program can be represented in HOCL by encoding its reaction rules by n-shot rules placed in the multiset. If needed, a rule can be removed by another rule in a reaction, thanks to the higher-order nature of the language (c.f. naming below).

Expressions. Expressions in HOCL consist in integer, boolean, string constants and associated operations. This extension is very standard and does not need further explanation. For example, a HOCL program computing the prime numbers smaller than 10 could be:

$$\textbf{let}\ sieve = \textbf{replace}\ x, y\ \textbf{by}\ x\ \textbf{if}\ x \text{ divide } y\ \textbf{in} \langle sieve, 2, 3, 4, 5, 6, 7, 8, 9, 10 \rangle$$

Types. The functional core of HOCL (the expressions) is statically typed using standard types. We do not describe the typing rules which are the same as any (first-order) statically typed functional language. The chemical style of programming has been designed to be very flexible. In particular, solutions contain usually molecules of different types (e.g., reactions, integers, etc.). Types are particularly useful in patterns where they serve to select values. The type is specified in a pattern by the operator '::'. We make use of type inference to circumvent type annotations in patterns. For instance, we may write $\gamma(x)\lfloor V \rfloor.x+1$ instead of $\gamma(x{::}\text{Int})\lfloor V \rfloor.x + 1$ since the type of x can be statically inferred.

Pairs and Tuples. This extension, denoted here by $A_1{:}A_2$, is very standard. Note that the elements of a pair are atoms and not multisets. Pairs of multisets would play a role similar to solutions by providing a way of isolating compound molecules from each other.

Empty Solutions. The notion of empty solution in HOCL comes from the pattern ω which can match any molecules even the "empty one" (introduced below). This pattern is very convenient to extract elements from a solution. For example, the following reaction extracts 1's from its solution argument.

$$rmunit = \textbf{replace} \langle x, \omega \rangle\ \textbf{by} \langle \omega \rangle\ \textbf{if}\ x = 1$$

The pattern ω matches the rest of the solution which is returned as result. If the solution contains only a 1 then ω matches the empty molecule and the empty solution is returned:

$$rmunit, \langle 2, 1, 3 \rangle \longrightarrow \langle 2, 3 \rangle \qquad \text{and} \qquad rmunit, \langle 1 \rangle \longrightarrow \langle \rangle$$

When a molecule M is decomposed into M_1, M_2 to match a pattern P_1, P_2, one of M_1 or M_2 can now be the empty molecule $\emptyset$. Reaction rules involving ω patterns need a special treatment. Consider, for example, the reaction

$$(\mathbf{replace}\langle x, \omega\rangle \mathbf{by}\, \omega), \langle 1\rangle, 2$$

With the usual reduction rules, this molecule would reduce to $\emptyset, 2$ which is not a legal molecule.

Naming. Reaction rules can be named (or tagged) using the syntax $name = x$, where *name* is a constant, and x the variable that will match the unnamed rule. Note that if others atoms can be named using pairs (e.g., *name*:a), it would not be appropriate to use pairs to tag rules since they would not be able to react with other molecules anymore. Names are used to match and extract specific reactions. We assume that when the **let** operator names a reaction. For example, in the following example, the reaction incrementing the integer is named *succ*. After an arbitrary number of increments, the reaction *stop* removes *succ* from the solution:

$$\begin{array}{l}\mathbf{let}\ succ = \mathbf{replace}\, x\, \mathbf{by}\, x+1\, \mathbf{in}\\ \mathbf{let}\ stop = \mathbf{one}\ succ = x, \omega\, \mathbf{by}\, \omega\, \mathbf{in}\\ \langle 1, succ, stop\rangle\end{array}$$

This example also illustrates non-determinism in HOCL since the resulting solution may be any integer.

Example of a distributed versions system (DVS). As a more involved example, we consider several persons editing concurrently a document made out of a set of files. These editors are distributed over a network and each one works on one node of that network. Each node is independent from the others. Each editor makes his own modifications in the files and commits them locally on his node. So each editor keeps a local version (and its history) of these files. That version consists in the start files and several ordered patches applied to them: this history is called a *branch*. From time to time, two or more editors merge their branches so that an editor propagates (pushes) its modifications to others and/or get changes from other editors.

The following example is inspired from Monotone, a distributed version control system (`http://venge.net/monotone/`). Versions are identified by a hash which is used to check whether two branches are identical (denoted by $b_1 \neq b_2$). The system can also identify modifications applied to a branch b_1 that have not been taken into account in another branch b_2 (denoted by $b_1 \not\subset b_2$). The system provides also the function $Merge(b_1, b_2)$ which returns a branch that contains all modifications from two given branches b_1 and b_2. If a conflict occurs, the initiator of the merge must resolve it. For simplicity sake, we assume in this example that the function *Merge* always succeeds: either there is not any conflict, or if any conflict occurs it is solved by an editor.

An editor can express his dependency on modifications made by other editors. If the editor on node N_i depends on modifications made by editor on

$$
\begin{array}{l}
\text{dvs} = \\
\quad \textbf{let } \mathit{edit} = \textbf{replace } b \textbf{ by } \mathit{Edit}(b) \textbf{ in} \\
\quad \textbf{let } \mathit{push} = \textbf{replace } b_1, b_2 \\
\qquad\qquad\qquad \textbf{by } b_1, \mathit{Merge}(b_1, b_2) \\
\qquad\qquad\qquad \textbf{if } \mathit{Serve}(b_1, b_2) \wedge b_1 \not\subset b_2 \\
\quad \textbf{in} \\
\quad \textbf{let } \mathit{sync} = \textbf{replace } b_1, b_2 \\
\qquad\qquad\qquad \textbf{by } \mathit{Merge}(b_1, b_2), \mathit{Merge}(b_2, b_1) \\
\qquad\qquad\qquad \textbf{if } \mathit{Serve}(b_1, b_2) \wedge \mathit{Serve}(b_2, b_1) \wedge b_1 \neq b_2 \\
\quad \textbf{in} \\
\quad \textbf{let } \mathit{crash} = \textbf{replace } b_1 \textbf{ by } \mathit{Start} \textbf{ if } \mathit{Crash}(b_1) \textbf{ in} \\
\quad \textbf{let } \mathit{freeze} = \textbf{replace } (\mathit{edit} = e), x \textbf{ by } x \textbf{ in} \\
\quad \textbf{let } \mathit{newVersion} = \textbf{replace } \langle b_1, x \rangle \textbf{ by } \mathit{NewRelease}(b_1), \langle b_1, \mathit{edit}, x \rangle \textbf{ in} \\
\quad \langle \langle B_1, \ldots, B_n, \mathit{edit}, \mathit{push}, \mathit{sync}, \mathit{crash}, \mathit{freeze} \rangle, \mathit{newVersion} \rangle
\end{array}
$$

Fig. 3. Distributed Versions System

node N_j then the boolean function $\mathit{Serve}(b_i, b_j)$ will be true. In other words, modifications present in the branch b_i should be propagated to the branch b_j. They may be both dependent on each other. Since any branch can merge with any other branch, editors have to organize themselves so that all modifications from all editors are taken into account sooner or later. For example, the *Serve* function may induce a tree where modifications may be propagated from the root to the leaves and vice versa. Or they may be organized as a ring, or any other structure. Regularly, a freeze (snapshot) of the document is made to release a new version to users. This is performed by a call to the function *NewRelease*.

The overall system is described the program of Fig. 3. It consists in a solution containing all branches b_i. The reaction rule *edit* represents the edition of any branch. It adds a modification to a branch, a call to the function *Edit*. Reaction rules *push* and *sync* merge branches: *push* propagates modifications in one way, and *sync* synchronizes two branches. If a node crashes ($\mathit{Crash}(b_i)$), the editor loses the corresponding branch. The reaction rule *crash* resets the corresponding branch to an empty branch (*Start*). At any time, the reaction *freeze* can initiate a snapshot of the document by removing the edition rule *edit* to stop any modification. When the solution becomes inert, all branches linked by a *Serve* relation are up to date and the reaction *newVersion* can occur. It uses a branch that has all the modifications (it depends on the relations *Serve*) to release a new version (a call to *NewRelease*) and regenerates the system by adding the rule *edit* to allow new modifications for the next release. Figure 4 gives a possible state reached by a system with 5 editors. The edition is pending and two releases have been made (*Version1* and *Version2*).

This example illustrates several properties of HOCL :

– The execution is *non-deterministic*. Any two branches may react to merge their differences (if at least one of them serves the other). Merges (reactions *push* and *sync*) may not occur each time a modification is made on a branch.

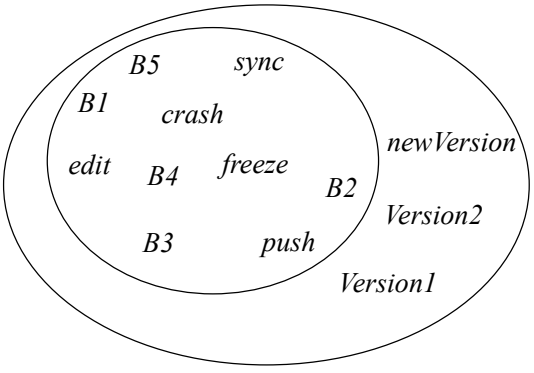


Fig. 4. A possible state of the DVS

In fact, editions and merges are asynchronous: several editions may occur before a merge.

- The execution is potentially *parallel*. Several editions may occur at the same time and several merges may happen at the same time if they deal with disjoint branches.
- The system is *autonomic* in that it is self-repairing. If a crash occurs, we lose a branch, but a simple *push* or *sync* with another branch allows to recover all modifications that have been propagated (however the editor loses all his local non-propagated modifications). Other autonomic properties may be included in a chemical program. The interested reader is referred to [3] for more details on autonomic chemical programs.
- The specification is *higher-order* and manipulates reaction rules to express coordination. The *freeze* reaction removes the *edit* rule to stop edition. The *newVersion* rule waits for inertia to call *NewRelease* which illustrates a basic sequentiality coordination. The *newVersion* rule relaunches also the system by re-generating the solution with the rule *edit*.

3.3 Multiplets, Infinite and Hybrid Multisets

Another generalization is to extend the class of multisets to infinite multisets and elements with a negative multiplicity. The extension amounts to introducing operations to explicitly manipulate the finite or infinite, positive or negative, multiplicity of elements.

Multiplets. A *multiplet* is a finite multiset of identical elements. This notion relies on an equality relation between elements. Considering multiplets of reaction rules would cause semantic problems as it would require an equality relation between programs; whereas multiplets of solutions would pose implementations issues. In this paper, we limit ourselves to multiplets of basic values (integers, booleans, strings). Multiplets are defined and matched using an exponential notation. If v is a basic value then v^k $(k > 0)$ denotes a multiplet of k elements

let $choose = \gamma\langle x{::}String, \omega\rangle.x$ **in**
let $wheel = \langle$"cherry", "lemon", "bell", "bar", "plum", "orange", "melon", "seven"$\rangle$ **in**
let $win = \gamma(x{::}String)^3$."Jackpot!" **in**
$\langle wheel, wheel, wheel, choose, choose, choose, win\rangle$

Fig. 5. The Jackpot! program

v. Similarly, if x is a variable of a basic type ($x{::}B$), x^k denotes a multiplet of k elements. In order to match multiplets, the language of patterns is extended likewise. A pattern, P^k matches any multiplet of k identical elements matching P. For example, the reaction replacing four 1's by four 2's can be specified as

$$\gamma(x^4)\lfloor x = 1\rfloor.2^4 \text{ or equivalently } \gamma(x,x,x,x)\lfloor x = 1\rfloor.2,2,2,2$$

Another elementary example is the n-shot reaction rule computing the root set of a multiplet by removing repeatedly pairs of identical elements:

$$toSet1 = \textbf{replace}\, x^2 \,\textbf{by}\, x$$

In the Jackpot! (see Fig. 5), the rules *choose* pick up non-deterministically an element from the solutions representing the three wheels of a slot machine. The *win* rule checks if the three drawn symbols are identical, i.e., if it can match a multiplet of size 3.

Variable-Sized Multiplets. A first generalization of multiplets is to allow variables in the exponentiation of constants or patterns. The size of a multiplet becomes dynamic.

Let v be a basic constant and V an integer expression, then v^V denotes a multiplet. If the normal form of V is the integer k then $v^V \equiv v^k$. We assume in this section that $k > 0$. If $k = 0$ the multiplet is empty and is treated in much the same way as a ω-variable which has matched the empty molecule (c.f. Section 3.2). The case of a negative exponent is dealt with in Section 3.3.

A pattern P^x matches any strictly positive number of identical basic values. For example, the n-shot rule computing the root set of a multiplet of the previous section can be expressed using variable sized multiplet matching:

$$toSet2 = \textbf{replace}\, x^n \,\textbf{by}\, x \,\textbf{if}\, n > 1$$

Whereas the previous version ($toSet1$) eliminated duplicates two by two, the rule $toSet2$ eliminates a variable number of (potentially greater than 2) duplicates at each step.

Infinite Multiplets. Another generalization consists in infinite multiplets. Let v be a basic value or a variable with a basic type, then v^∞ denotes an infinite multiplet made of an infinity of copies of v. We do not introduce patterns of the form P^∞ to match an infinity of identical elements. Indeed, extracting an infinity of elements from an infinity would not be well defined. Instead we introduce

a pattern matching all occurrences of a constant in the solution. Using such patterns, infinite multiplets can be manipulated as a single atomic molecule.

The pattern $P^{\overline{x}}$ matches all identical atoms occurring in the enclosing solution. The substitution returned by a successful match maps the variable x to the finite or infinite multiplicity of the matched value.

For example, the n-shot reaction computing the root set of a multiplet of the previous sections can now be expressed as follows:

$$toSet3 = \textbf{replace}\, x^{\overline{n}}\, \textbf{by}\, x\, \textbf{if}\, n > 1$$

All duplicates of an element are removed in one reaction rule. For example, the solution $\langle a^{10}, b^4, toSet3\rangle$ is rewritten in two steps:

$$\langle a^{10}, b^4, toSet3\rangle \longrightarrow \langle a, b^4, toSet3\rangle \longrightarrow \langle a, b, toSet3\rangle$$

As another example, consider the traditional quicksort program where a set of integers has to be compared with a predefined pivot. In order to distinguish the pivot from the other integers, we assume that the pivot has a special type $Pivot$ (e.g., a type synonym of Int). In the following solution all integers lower or equal to the pivot are removed. We consider the pivot as a infinite multiplet of an integer of type $Pivot$ (5^∞ here):

$$\langle 5^\infty, 8, 3, 6, 4, 5, 3, \textbf{replace}(p{::}Pivot), x, \omega\, \textbf{by}\, \omega\, \textbf{if}\, x \leq p\rangle$$

As the number of pivots is infinite, all possible reactions may be carried out independently. This is a way of expressing the fact that the pivot is a read only element and as such can be accessed concurrently. The use of read only elements in chemical specifications has been proposed in [38].

Negative Multiplicities. Hybrid multisets [39,8] are a generalization of multisets where the multiplicity of elements can be negative. A molecule v^{-1} can be viewed as a piece of "antimatter" or an anti-v. Positive and negative multiplets of the same value cannot cohabit in the same solution, they merge into one multiplet whose exponent is the sum of their exponent. Assuming a representation of negative values v^{-1}, a negative multiplet v^{-k} is defined as k occurrences of v^{-1}. The pattern P^{-1} is defined as matching (the representation of) v^{-1} such that P match v. The pattern P^{-k} is defined as k occurrences of P^{-1}. The intended semantics enforces that v and v^{-1} cannot be in a solution at the same time. When negative multiplicities are allowed, the negative and positive multiplets of the identical elements must be merged after each reaction before proceeding with other reactions. In other words, reactions become global rewritings w.r.t. their solution.

As an example of use of negative multiplicities, rational numbers $\frac{p}{q}$ are represented by a molecule which contains the prime factorization of p and q but with negative multiplicities for the latter. For example, $\frac{20}{9}$ is represented by the molecule $\langle 2^2, 5, 3^{-2}\rangle$. The product of rational numbers is computed simply by putting them in the same solution. For example, the product $\frac{20}{9} * \frac{15}{8}$ is performed by merging their representations:

$$\langle 2^2, 5, 3^{-2}\rangle, \langle 3, 5, 2^{-3}\rangle, \gamma(\langle f\rangle, \langle g\rangle).\langle f, g\rangle \longrightarrow \langle 5^2, 3^{-1}, 2^{-1}\rangle$$

Infinite negative multiplets can be used to filter out all occurrences of an element (present or to come) within a solution. Let pi be the reaction computing the product of a multiset of integers. Then, the integer 1, being the neutral element of the product, can be deleted prior to performing pi. The pi operator may be encoded by:

$$pi = \gamma\langle x\rangle.\langle 1^{-\infty}, x, (\textbf{replace}\, x, y\, \textbf{by}\, x * y)\rangle$$

Before considering any product, all 1's are annihilated, for example:

$$\langle 2^2, 9, 1^3, 5, 6\rangle, pi \longrightarrow \langle 1^{-\infty}, 2^2, 9, 5, 6, (\textbf{replace}\, x, y\, \textbf{by}\, x * y)\rangle \longrightarrow \ldots$$

Note that (by type inference) the pattern x, y exactly matches two integers (and not anti-1's). Furthermore, since $1^{-\infty}$ is in the solution, x and y will never match a 1. After stabilization, $1^{-\infty}$ must be replaced by 1 (in case that the solution contained only 1's) and then the reaction rule can be removed.

Other examples that come to mind include the specifications of a garbage collector that destroys useless molecules by generating their negative counterpart, or an anti-virus that generates $v^{-\infty}$ each time it identifies a virus v. The negative multiplet will remove all occurrences (present or future) of the corresponding virus from the solution.

[40] gives the operational semantics of multiplets, variable sized, infinite and negative multiplets.

4 Applications and Case Studies

We illustrate the expressive power of our higher-order model by encoding several well known coordination mechanisms in HOCL. We also describe the application of HOCL to the specification of self-organizing systems.

4.1 Chemical Coordination

Classical coordination can be expressed in a chemical manner [41]. The chemical reaction model already includes some basic coordination mechanisms that can be used to express more elaborate coordination mechanisms.

HOCL is a programming language that already provides some primitive coordination structures: namely, parallel execution, mutual exclusion, the atomic capture and the serialization and parallelization of computations.

Parallel Execution. When two reactions involve two separate multisets of reactives, both reactions can occur at the same time. For example, when computing the sum of a multiset of integers:

$$\langle 42, 6, 14, 5, 2, 8, 5, 42, 89, \text{add} = \textbf{replace}\, x, y\, \textbf{by}\, x + y\rangle$$

several reactions involving the rule add may occur at the same time. Parallel execution relies on a fundamental property of HOCL : mutual exclusion.

Mutual Exclusion. The mutual exclusion property states that a molecule cannot take part to several reactions at the same time. For example, several reactions can occur at the same time in the previous solution (e.g., (42,89) at the same time as (5,5), etc.). Without mutual exclusion, the same number could occur in several reactions at the same time. In this case, our previous program would not represent the sum of a multiset since, for example, 89 would be allowed to react with 2 and 6 and be replaced by 91 and 95.

Atomic Capture. Another fundamental property of HOCL is the atomic capture. A reaction rule takes all its arguments atomically. Either all the required arguments are present or no reaction occurs. If all the required arguments are present, none of them may take part in another reaction at the same time. Atomic capture is useful to express non blocking programs. For example, the famous dining philosophers problem can be expressed in HOCL as follows:

$$\text{eat} = \textbf{replace}\ \text{Fork}{:}f_1,\ \text{Fork}{:}f_2\ \textbf{by}\ \text{Phi}{:}f_1\ \textbf{if}\ f_2 = f_1 + 1 \bmod N$$

$$\text{think} = \textbf{replace}\ \text{Phi}{:}f\ \textbf{by}\ \text{Fork}{:}f,\ \text{Fork}{:}(f + 1 \bmod N)\ \textbf{if true}$$

Initially the multiset contains N forks (i.e., N pairs Fork:1, ..., Fork:N) and the two n-shot reaction rules eat and think. The eat rule looks for two adjacent forks Fork:f_1 and Fork:f_2 with $f_2 = f_1 + 1 \bmod N$ and "produces" the eating philosopher Phi:f_1. This reaction relies on the atomic capture property: the two forks are taken simultaneously (atomicity) and this prevents deadlocks. The think rule "transforms" an eating philosopher into two available forks. This rule models the fact that any eating philosopher can be stopped non deterministically at anytime.

Serialization. A key motivation of chemical models in general, and HOCL in particular, is to be able to express programs without any artificial sequentiality (i.e., sequentiality that is not imposed by the logic of the algorithm). However, even within this highly unconstrained and parallel setting, sequencing of actions can be expressed. Sequencing relies on the fact that a rule needing to access a sub-solution has to wait for its inertia. The reaction rule will react after (in sequence) all the reactions inside the sub-solution have completed. The HOCL program that computes all the primes lower than a given integer N can be expressed by a sequence of actions that first computes the integers from 1 to N and then applies the rule sieve:

$$\langle\langle \text{iota}, \overline{N}\rangle, \text{thensieve}\rangle$$

where

$$\begin{aligned} \text{thensieve} &= \textbf{one}\langle \text{iota} = r, \overline{x}, \omega\rangle\ \textbf{by}\ \text{sieve}, \omega \\ \text{iota} &= \textbf{replace}\ \overline{x}\ \textbf{by}\ x, \overline{x-1}\ \textbf{if}\ x > 1 \\ \text{sieve} &= \textbf{replace}\ x, y\ \textbf{by}\ x\ \textbf{if}\ x \text{ div } y \end{aligned}$$

The rule iota generates the integers from N to 1 using the notation $\overline{x}$ to denote a distinguished (e.g., tagged) integer. The one-shot rule thensieve waits for the

inertia of the sub-solution. When it is inert, the generated integers are extracted and put next to the rule sieve (iota and the tagged integer $\overline{1}$ are removed). The wait for the inertia has serialized the iota and sieve operations.

Most of the existing chemical languages share these basic features. They all have conditional reactions with atomic capture of elements. On the other hand, they usually do not address fairness issues.

[41] develops more advanced mechanisms such as rendez-vous, shared variables, Linda primitives, Petri nets and Kahn Process Networks.

4.2 Self-organization

Principle. N-shot abstractions are well fitted to express self-management properties. For example, computing the prime numbers up to 5 can be expressed as:

$$\langle \text{sieve}, 2, 3, 4, 5\rangle \longrightarrow \langle \text{sieve}, 2, 3, 5\rangle$$

where *sieve* is the reaction rule that removes an integer if it finds a divisor. The molecule "sieve" is part of the result (stable state). If new integers are added (perturbation), reactions may start again until a new inert solution is reached (new stable state). For example, if we need the prime numbers up to 10, we may just add integers to the previous inert solution:

$$\langle \text{sieve}, 2, 3, 4, 5\rangle,\ \gamma\langle x\rangle.\langle x, 6, 7, 8, 9, 10\rangle$$

and the solution will re-stabilize to $\langle \text{sieve}, 2, 3, 5, 7\rangle$. The molecule "sieve" can be seen as an invariant: it describes the valid inert states (here, set of prime numbers). In the next section, we make use of this property to add several self-management features to a mail system.

A Self-Organizing Mail System Example. In [3], we describe an autonomic mail system within the higher-order chemical framework. This example illustrates the adequacy of the chemical paradigm to the description of autonomic systems.

Several self-management features for the mail system have been developed: self-organization, self-healing (by providing emergency mail servers), self-optimization (by enabling the emergency server and load-balancing messages between it and the main server) and self-configuration (managing mobile clients).

The mail system is described as a chemical solution where some sub-solutions described mailboxes, mail servers and network. Some rules are in charge of forwarding messages to their recipient by moving messages from sub-solutions to other sub-solutions. Thus sending and receiving a message is performed by self-organization. Adding a message to be sent in the system perturbs the system. A message sent but not received create a unbalance detected by rules which perform the correct operation to reach a balance where the message sent is delivered.

[3] shows also that other self-management properties can be programmed with HOCL. For example, self-healing can be set by using an emergency mail servers. Some rules are in charge of detecting that a main server has failed, then they

add some new rules in the system which use an emergency server (changing dynamically the behavior of the system). In a similar way, self-optimization can be expressed by some rules which enable the emergency server and load-balance messages between it and the main server, and self-configuration by some rules that manage mobile clients.

Each self-management property is represented by a set of rules. Adding a self-property to the mail system consists simply in adding the corresponding rules to the main solution representing the system.

This description should be regarded as a high-level parallel and modular specification. It allows to design and reason about autonomic systems at an appropriate level of abstraction. The resulting programs are quite elegant; they rely essentially on the higher-order and chemical nature of Gamma. A direct implementation of the chemical specifications is likely to be quite inefficient and further refinements are needed; this is another exciting research direction.

5 Prospects

The very high-level nature of HOCL makes it stand somewhere between a specification language and an implementation oriented language. As a consequence, direct and naive implementations of chemical programs usually lead to very inefficient systems. Our current research focuses on making chemical programming effective. Two directions are being explored:

- The refinement of chemical programs. The goal is to express separately implementation choices so that the chemical program can be refined into a lower level and more efficient version.
- Domain-specific chemical languages. The objective is to propose specialized, restricted and effective chemical languages for domains such as autonomic or grid computing.

5.1 Refinement of Chemical Programs

A chemical program focuses on the base functionalities and usually disregards optimizations and implementation tricks. This is reflected by the unconstrained data structures (multisets) and evaluation strategies (non deterministic chaotic reactions).

Consider, as a simple illustration, the maximum segment sum problem [42] expressed as a reaction rule. The input parameter is a sequence of integers. A segment is a subsequence of consecutive elements and the sum of a segment is the sum of its values. The program returns the maximum segment sum of the input sequence. Each element of the segment is represented by a triplet (i, v, s) where i denotes the index of the element in the sequence, v the value of the element, and s serves to computes the maximum segment sum. The following reaction computes sums for each segment.

$$\begin{aligned} \textbf{replace}\ & (xi, xv, xs), (yi, yv, ys) \\ \textbf{by}\ & (xi, xv, xs), (yi, yv, xs + yv) \\ \textbf{if}\ & yi = xi + 1 \wedge xs + yv > ys \end{aligned}$$

Initially, the reaction is in a multiset of triples of the form (i, v, v). A sequence $v_1{:}\ldots{:}v_n$ is represented by the multiset $(1, v_1, v_1), \ldots, (n, v_n, v_n)$. For example, the sequence 3:1: − 1:2 is represented as $(1, 3, 3), (2, 1, 1), (3, -1, -1), (4, 2, 2)$. When the multiset is stable, in each triplet (i, v, s) s denotes the maximum segment sum ending at index i. It is then sufficient to select the triplet (i, v, s) with the greatest s to solve the classic problem. The previous sequence will stabilize with the following triplets

$$(1, 3, 3), (2, 1, 4), (3, -1, 3), (4, 2, 5)$$

and the maximum segment sum is 5 and ends at index 4. No execution order is specified and it is easy the see that the worst case complexity is $\mathcal{O}(n^3)$ with n the size of the sequence (e.g., using a strategy choosing the first element (i, v, s) in decreasing order of i). Remember that the standard imperative algorithm of maximum segment sum has a linear time complexity.

The higher complexity of chemical programs has two main reasons:

- the selection of tuples is largely unspecified. In the example, only successive elements of the sequence can react. The successor relation is not explicit in the multiset representation; the reaction cannot directly pick two successive elements without testing the first field of the triplets;
- the detection of termination. Consider a multiset of size N with a unique k-ary reaction returning k elements (i.e., maintaining the cardinal of the multiset). All possible k-uples should be tested to decide termination, a $\mathcal{O}(N^k)$ operation.

In [25], Creveuil studied transformations of Gamma programs aimed at improving these two points. The optimizations depended on properties which had to be checked manually. This approach can be seen as a methodology to refine Gamma programs manually.

Our aim is to study and propose linguistic tools to help the programmer to express implementation issues and program refinement. Several techniques and research directions are worth studying. We review them in turn.

Data Representation. The lack of support for structuring data in HOCL should not be surprising since the motivation behind chemical programming is to be able to describe programs exhibiting as few ordering constraints as possible. An unfortunate consequence however is that the programmer sometimes has to resort to artificial encodings to express his algorithm. For instance, the maximum segment sum algorithm shown above is expressed in terms of triplets with indexes. The lack of structuring facility is detrimental both for reasoning about programs and for implementing them.

In [22], we proposed a solution to this problem for Gamma without jeopardizing the basic qualities of the language. We introduce structured multiset which is specified by a type (a graph grammar) expressing a form of neighborhood between the molecules of the solution. It allows the programmer to define his own types and have his programs type checked. Graph grammars can model

most classic pointer data structures such as lists, trees, doubly-linked circular lists, etc. The *List* type is defined based on a binary relation **next** x y stating that the address y is the successor of the address x. For example, the previous sequence of triplets is represented as **next** a b, **next** b c, **next** c d where a, b, c, d are addresses whose value fields are $a.v = a.s = 3$, $b.v = b.s = 1$, $c.v = c.s = -1$, $d.v = d.s = 2$. The maximum segment sum program can be expressed as

$$\begin{aligned} \textbf{replace}\ & \textbf{next}\ x\ y \\ \textbf{by}\ & \textbf{next}\ x\ y, y.s := x.s + y.s \\ \textbf{if}\ & x.s + y.v > y.s \end{aligned}$$

Here only successors are compared and the complexity drops to $\mathcal{O}(n^2)$.

Another approach is followed in the MGS project [43]. In MGS, data structures are formalized as a chain complex (a discrete topological space labeled by values) [44]. In this setting, a multiset is a space where all elements are neighbors whereas a sequence is a space where neighbors are the consecutive elements of the sequence. The abstract topological point of view enhances chemical programming and enables the unification in a same programming language of several computational models.

By restricting the possible reactions, data structures also make the implementation more efficient. It is still unclear how such approaches can be adapted in a higher-order setting. In any case, data structures and a type system well suited to HOCL remain to be studied and developed.

Evaluation Strategies. For the same reasons as data structures (i.e., introducing as few contraints as possible), there is no support for expressing specific reduction strategies in Gamma or HOCL. Controlling the evaluation strategy is particularly useful to remove the cost of checking termination. As above, this can be achieved using types and data structures. In [22], we refined the *List* type with two extra relations used to distinguish between inert and active elements. A specific strategy can be expressed subsequently by taking these new relations into account within reactions. With these new relations, the maximum segment sum program can be expressed as a one-pass walk through the list. Of course, it must be proved that the strategy is correct (i.e., find the same normal forms as the original program).

Another approach is followed by the rewriting community using strategy languages. For example, Elan [45] proposes a language to express complex user-defined strategies using primitives such as sequential composition, iteration and (non-)deterministic choices. Rewriting rules and strategies are expressed at the same level in their own dedicated language.

Aspects of Implementation. Ideally, the specification of the base functionality (as a chemical program) would remain separate from the specification of implementation choices. In particular, data structures and evaluation strategies should be expressed separately and then used to automatically refine the chemical program. This idea is similar to Aspect Oriented Programming (AOP) [46] that isolates aspects (such as security, synchronization or error handling) whose

implementation would otherwise yield tangled code. In AOP, such aspects are specified separately and integrated into the program by an automatic transformation process called weaving. In the chemical context, the idea would be to consider data representation and evaluation strategy aspects. Weaving would take the chemical program plus the aspects and would produce a lower-level program integrating these implementation choices.

5.2 Domain Specific Chemical Languages

A second approach to guarantee efficient implementations is to restrict and specialize HOCL. Such an approach has been explored for the Gamma language where five schemes (basic reactions) have been identified [10]. Many applications can be expressed using only those schemes. An important benefit is that these schemes can be implemented more efficiently.

Autonomic computing is a good candidate for the design of a domain specific chemical language. Autonomicity (e.g. self-healing, self-protection, self-optimization, etc.) is naturally expressed as reaction rules maintaining an invariant [47]. In each case, the corresponding behavior can be seen as the stabilization of the system after a transient perturbation. It is very likely that the expression of self-$*$ properties does not need the full expressive power of HOCL. Also, reactions always proceed from a stable multiset to which a small collection of elements is added (the perturbation). It should be possible to specialize/optimize the implementation for this peculiar rewriting scheme (only reactions involving the new elements have to be considered).

We are currently investigating another application domain: grid computing. Grids are systems made of a huge number of independent resources connected through a network. Every resource supplies some of its capabilities (memory, computation power and network) to the grid, so that the power of a grid is the sum of the powers of all the resources it is made of. A grid operating system must take into account the characteristics of the resources: their huge number, their heterogeneity, their non-reliability and their dynamicity.

Chemical programming can be a good candidate for grid programming and coordination. The chemical paradigm suits the requirements of a grid programming language for at least two reasons: (1) the locality principle, that suggests to write reactions involving few molecules compared to the size of the solution, increases the potential parallelism and allows computation to be performed without a global view, and (2) the self-organization property of the chemical coordination is the best coordination mechanism for such a dynamic system.

In [4], the objective is to design a Desktop Grid that is based on a peer-to-peer approach in which there is not a unique server any more to coordinate the execution of computations on unused PCs. The coordination of computations is described by a high level description of the coordination based on chemical programs. However, we do not argue here that all computations must be expressed using chemical rules since it will lead to a very ineffective Desktop Grid system. Basic computations have to be implemented using existing programming languages, such as Java to handle resource heterogeneity. The multiset is

implemented in a distributed way by having each PC provide part of their memory resources to store data, chemical rules and basic computations. Therefore, when an unused PC is willing to join a Chemical Desktop Grid, it accesses the multiset, downloads the basic computation, such as a Java bytecode, and starts executing the chemical rules.

Using the same approach as in [47], a "chemical" desktop grid can be described by a HOCL program. A chemical grid is viewed as a chemical solution of resources where reaction rules describe their coordination. A program is written in two steps that separate two concerns. The first concern is to describe the computation at the functional level (coordination of functions) and the second concern is the computation related to the execution of the previous functional computation in a distributed system (coordination of resources). [4] goes further and provides an example of a ray-tracer described by a HOCL program that it then refined into a HOCL program simulating its execution as a chemical grid.

A future research about the application of the chemical paradigm to grid programming is the implementation. An efficient implementation on grid would be based on distributed versions of previous parallel implementations of Gamma [14,15,18]. Another technique may be based on using chemical data-structures, like in Structured Gamma, to reinforce the locality in finding reacting molecules. The provided grid chemical language would propose some abstract mechanism that may be implemented in an efficient way on a distributed system.

In related works, the chemical programming model has also been used to describe workflows and their execution in a distributed system [48,49]. Nature is a great source of inspiration to tackle the problems posed by grids. For example, Organic Grids [50] are based on self-organizing autonomous agents inspired by biological systems to discover new resources in a grid.

6 Conclusion

To summarize the main progress achieved in recent years concerning the chemical reaction paradigm, one can emphasize:

1. the use of a very general version of multisets with elements possessing various kinds of (finite or infinite) multiplicities;
2. the introduction of a higher order model of computation (HOCL) dealing with such general data structures.

Basically, the chemical paradigm (as introduced in HOCL) offers four basic properties: mutual exclusion, atomic capture, parallelization and serialization. These properties have been exploited in order to give a chemical expression of well known coordination schemes such as CSP rendezvous, shared variables, Linda's primitives, Petri nets and Kahn Process Networks in HOCL. They have also been used to specify autonomic systems at a high-level [41].

We believe that the chemical programming approach is a good candidate to program large scale distributed systems such as service infrastructures and Grids. More precisely, we are exploring the use of HOCL to perform the orchestration

of a large number of services for applications that require self-adaptation and fault-tolerance. The objective is to be able to adapt the workflow of an existing application depending of the occurrence of logical and physical failures that might happened within a Grid or Service infrastructures. HOCL provide a coherent framework for self-adaptation by allowing rules to react on rules stored in the multiset; both the workflow and its adaptation are expressed by a set of HOCL rules.

As a final conclusion, borrowing the final sentence of the "fifteen year after" survey [7], we hope that this survey has shown that the chemical reaction model is a particularly simple and fruitful paradigm; no doubt that new surprising and exciting developments are yet to come.

References

1. Papazoglou, M.P., Georgakopoulos, D.: Service-oriented computing. Communications of the ACM 46(10) (2003)
2. Plummer, D.C., Cearley, D.W., Smith, D.M.: Cloud computing confusion leads to opportunity (June 2008), http://www.gartner.com/it/products/research/cloud_computing/cloud_computing.jsp
3. Banâtre, J.P., Fradet, P., Radenac, Y.: Chemical specification of autonomic systems. In: Proc. of the 13th Int. Conf. on Intelligent and Adaptive Systems and Software Engineering (IASSE 2004) (2004)
4. Banâtre, J.P., Le Scouarnec, N., Priol, T., Radenac, Y.: Towards "chemical" desktop grids. In: Proceedings of the 3rd IEEE International Conference on e-Science and Grid Computing (e-Science 2007). IEEE Computer Society Press, Los Alamitos (2007)
5. Banâtre, J.P., Priol, T., Radenac, Y.: Service orchestration using the chemical metaphor. In: Brinkschulte, U., Givargis, T., Russo, S. (eds.) SEUS 2008. LNCS, vol. 5287. Springer, Heidelberg (2008)
6. Banâtre, J.P., Le Métayer, D.: Programming by multiset transformation. Communications of the ACM (CACM) 36(1), 98–111 (1993)
7. Banâtre, J.P., Fradet, P., Le Métayer, D.: Gamma and the chemical reaction model: Fifteen years after. In: Calude, C.S., Pun, G., Rozenberg, G., Salomaa, A. (eds.) Multiset Processing. LNCS, vol. 2235, pp. 17–44. Springer, Heidelberg (2001)
8. Loeb, D.: Sets with a negative number of elements. Advances in Mathematics 91, 64–74 (1992)
9. Banâtre, J.P., Le Métayer, D.: A new computational model and its discipline of programming. Technical Report RR0566, INRIA (September 1986)
10. Hankin, C., Le Métayer, D., Sands, D.: A parallel programming style and its algebra of programs. In: Reeve, M., Bode, A., Wolf, G. (eds.) PARLE 1993. LNCS, vol. 694, pp. 367–378. Springer, Heidelberg (1993)
11. Creveuil, C., Moguérou, G.: Développement systématique d'un algorithme de segmentation d'images à l'aide de Gamma. Techniques et Sciences Informatiques 10(2), 125–137 (1991)
12. McEvoy, H.: Gamma, chromatic typing and vegetation, pp. 368–387 (1996)
13. Ruiz Barradas, H.: Une approche à la dérivation formelle de systèmes en Gamma. PhD thesis, Université de Rennes 1, France (July 1993)

14. Banâtre, J.P., Coutant, A., Le Métayer, D.: A parallel machine for multiset transformation and its programming style. Future Gener. Comput. Syst. 4(2), 133–144 (1988)
15. Banâtre, J.P., Coutant, A., Le Métayer, D.: Parallel machines for multiset transformation and their programming style (1988)
16. Creveuil, C.: Implementation of gamma on the connection machine. In: Research Directions in High-Level Parallel Programming Languages, pp. 219–230 (1991)
17. Huan, L.P., Ng, K.W., Sun, Y.Q.: Implementing gamma on maspar mp-1, pp. 94–99 (1995)
18. Gladitz, K., Kuchen, H.: Parallel implementation of the gamma-operation on bags. In: Proc. of the PASCO (Parallel Symbolic Computation) conference, pp. 154–163 (1994)
19. Vieillot, M.: Synthèse de programmes Gamma en logique reconfigurable. Techniques et Sciences Informatiques 14, 567–584 (1995)
20. Hankin, C., Le Métayer, D., Sands, D.: A calculus of gamma programs. In: Proc. of the 5th International Workshop on Languages and Compilers for Parallel Computing, pp. 342–355. Springer, Heidelberg (1993)
21. Le Métayer, D.: Higher-order multiset programming. In: Proc. of the DIMACS workshop on specifications of parallel algorithms. Dimacs Series in Discrete Mathematics, vol. 18 (1994)
22. Fradet, P., Le Métayer, D.: Structured gamma. Science of Computer Programming 31(2–3), 263–289 (1998)
23. Dershowitz, N., Manna, Z.: Proving termination with multiset orderings. Communications of the ACM 22(8), 465–476 (1979)
24. Banâtre, J.P., Le Métayer, D.: The gamma model and its discipline of programming. Science of Computer Programming 15(1), 55–77 (1990)
25. Creveuil, C.: Techniques d'analyse et de mise en œuvre des programmes Gamma. PhD thesis, Université de Rennes 1, France (December 1991)
26. Chaudron, M.R.V., de Jong, E.D.: Towards a compositional method for coordinating gamma programs. In: COORDINATION, pp. 107–123 (1996)
27. McEvoy, H., Hartel, P.H.: Local linear logic for locality consciousness in multiset transformation. In: Swierstra, S.D. (ed.) PLILP 1995. LNCS, vol. 982, pp. 357–379. Springer, Heidelberg (1995)
28. Berry, G., Boudol, G.: The chemical abstract machine. Theoretical Computer Science 96, 217–248 (1992)
29. Boudol, G.: Some chemical abstract machines. In: A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium, London, UK, pp. 92–123. Springer, Heidelberg (1994)
30. Fradet, P., Le Métayer, D.: k Shape types. In: POPL 1997: Proc. of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 27–39. ACM, New York (1997)
31. Inverardi, P., Wolf, A.L.: Formal specification and analysis of software architectures using the chemical abstract machine model. IEEE Trans. Softw. Eng. 21(4), 373–386 (1995)
32. Le Métayer, D.: Describing software architecture styles using graph grammars. IEEE Trans. Softw. Eng. 24(7), 521–533 (1998)
33. Hoffmann, B.: Shapely hierarchical graph transformation. In: HCC, pp. 30–37 (2001)
34. Mentré, D., Métayer, D.L., Priol, T.: Formalization and verification of coherence protocols with the gamma framework. In: PDSE, pp. 105–113 (2000)

35. Ciancarini, P., Fogli, D., Gaspari, M.: A logic language based on gamma-like multiset rewriting. In: ELP, pp. 83–101 (1996)
36. Banâtre, J.P., Fradet, P., Radenac, Y.: Principles of chemical programming. In: Abdennadher, S., Ringeissen, C. (eds.) Proceedings of the 5th International Workshop on Rule-Based Programming (RULE 2004). ENTCS, vol. 124, pp. 133–147. Elsevier, Amsterdam (2005)
37. Păun, G.: Computing with membranes. Journal of Computer and System Sciences 61(1), 108–143 (2000)
38. Chaudron, M.: Schedules for multiset transformer programs. Technical Report tr94-36, Rijksuniversiteit Leiden (December 1994)
39. Blizard, W.: Negative membership. Notre Dame Journal of Formal Logic 31(3), 346–368 (Summer 1990)
40. Banâtre, J.P., Fradet, P., Radenac, Y.: Generalised multisets for chemical programming. Mathematical Structures in Computer Science 16(4), 557–580 (2006)
41. Banâtre, J.P., Fradet, P., Radenac, Y.: Classical coordination mechanisms in the chemical model. In: From semantics to computer science: essays in honor of Gilles Kahn. Cambridge University Press, Cambridge (2008)
42. Gries, D.: The maximum-segment-sum problem. Formal development programs and proofs, 33–36 (1990)
43. Giavitto, J.L., Michel, O.: MGS: a rule-based programming language for complex objects and collections. Electronic Notes in Theoretical Computer Science 59(4), 286–304 (2001)
44. Giavitto, J.L., Michel, O.: Data structure as topological spaces. In: UMC, pp. 137–150 (2002)
45. Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.E.: Elan from a rewriting logic point of view. Theor. Comput. Sci. 285(2), 155–185 (2002)
46. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241. Springer, Heidelberg (1997)
47. Banâtre, J.P., Fradet, P., Radenac, Y.: Programming self-organizing systems with the higher-order chemical language. International Journal of Unconventional Computing 3(3), 161–177 (2007)
48. Németh, Z., Pérez, C., Priol, T.: Workflow enactment based on a chemical metaphor. In: SEFM 2005: Proc. of the Third IEEE International Conference on Software Engineering and Formal Methods, pp. 127–136 (September 2005)
49. Németh, Z., Pérez, C., Priol, T.: Distributed workflow coordination: Molecules and reactions. In: The 9th International Workshop on Nature Inspired Distributed Computing, p. 241. IEEE, Los Alamitos (2006)
50. Chakravarti, A., Baumgartner, G., Lauria, M.: The organic grid: self-organizing computation on a peer-to-peer network. IEEE Transactions on Systems, Man and Cybernetics, Part A 35(3), 373–384 (2005)

Spatial Organization of the Chemical Paradigm and the Specification of Autonomic Systems

Jean-Louis Giavitto[1], Olivier Michel[1,2], and Antoine Spicher[2,3]

[1] IBISC FRE 3190 CNRS, Université d'Evry, Genopole,
523 place des Terrasses de l'Agora, 91000 Evry, France
{giavitto,michel}@ibisc.univ-evry.fr
http://mgs.ibisc.univ-evry.fr
[2] LACL EA 4213 Université Paris 12 (Paris Est),
61 Av. du Géneral de Gaulle, 94010, Créteil, France
{michel,spicher}@univ-paris12.fr
[3] LORIA UMR 7503 INRIA, CNRS, INPL, UHP, Nancy 2,
Campus Scientifique - BP 239, 54506 Vandoeuvre-lès-Nancy Cedex, France
Antoine.Spicher@loria.fr

Abstract. The *chemical paradigm* is an unconventional programming paradigm well fitted to the *high-level specification of parallel systems*. Based on the fixed point iterations of local rules, its use has been advocated for the programming of *autonomic* and *amorphous systems*. However, this model lacks an explicit handling of *spatial relationships*.

In this contribution, we first show how the chemical paradigm can be extended beyond multisets to arbitrary *topological collections*. Topological collections handle in a uniform way sophisticated data structures required in algorithmics as well as distributed data structures needed for the programming of autonomic or amorphous systems. Then we adapt a well-known result on multiset ordering to the more general case of topological collections. Well-founded ordering on topological collection can be used to prove the termination of the fixed point iteration of local rules.

1 Introduction

1.1 Gamma and the Chemical Paradigm

Introduced by the Gamma language, the chemical reaction metaphor [BCM88] describes computations in terms of reactions between molecules representing data, in a chemical solution represented as a multiset. A multiset is a generalization of a set that allows several occurrences of the same element. Computation proceeds by rewriting elements of a multiset according to conditions and transformation rules. The result of a chemical program is obtained when a stable solution is reached, i.e. when no reaction can take place anymore. For example, the reaction

$$convex\_hull = \textbf{replace } x, y, z, u \textbf{ by } x, y, z \textbf{ if } \mathrm{inside}(u; x, y, z)$$

M. Wirsing et al. (Eds.): Software-Intensive Systems, LNCS 5380, pp. 235–254, 2008.

replaces four points x, y, z and u by the first three points (i.e. u is removed) if u is inside the triangle x, y, z [BL90]. These replacements are repeated until a stable state is reached, that is to say, when no quadruple (x, y, z, u) can be found. The final stable solution contains exactly the elements defining the convex hull of the multiset of points specified in the initial solution.

1.2 Gamma and the Autonomic Computing Challenge

The goal of *Autonomic Computing* [Hor01] is to realize self-managing computer and software relying on properties of:

- *self-organization*: autonomous configuration of the components into a dynamic architecture dedicated to the satisfaction of the defined requirements;
- *self-healing*: autonomous detection and correction of hardware and software faults; and
- *self-optimization*: autonomous monitoring, control of resources and reconfiguration to ensure an optimal functioning.

The chemical paradigm has been claimed well suited to express autonomic properties: the reaction rules correspond to the local actions to be taken to react to a perturbation. Several convincing examples have been developed [BRF04].

We believe that the relevance of the chemical paradigm for the specification and the high-level programming of large autonomic and parallel/distributed systems comes from two fundamental characteristics:

1. the multiset data structure and the multiset rewriting device suitably represent the orderless interactions (reactions) between elements that occur in large parallel or open systems;
2. the computation of a stable state such that self-$*$ behaviors can been seen as the stabilization of the system on a fixed point after a transient perturbation.

However, these two general statements must be refined:

- The direct interactions of arbitrary elements in a system are not always allowed nor desirable. The system may exhibit some data organization and only "neighbor" elements may interact. The neighborhood relationship may represent physical (spatial distribution, localization of the resources) or logical constraints (inherent to the problem to be solved).
- A multiset stable w.r.t. the reactions represents a solution computed by the program or an admissible state of an autonomic system. This state is best characterized by global properties (e.g. the extremal points in a multiset of points in the convex hull computation) while the reactions represent local changes (e.g. the removal of one point fulfilling some conditions). Therefore, the real difficulty of chemical programming lies in the relation between the local changes and the desired global property.

In this paper, we present some concepts and tools in the field of algebraic topology that can be used to build more structured chemical solutions (section 2). For

the second point, we adapt a well-known result on multiset ordering that can be used to establish the convergence of local iterations of reactions (section 3). This result is a first step in the development of a toolbox of theoretical tools that can be used to link the local changes of elements to the global behavior of a system.

2 Introducing Space in the Chemical Paradigm

2.1 From Multisets to Sequences and Beyond

Multisets are a "loose organization" where more structured data require some encoding to be represented. For example, a sequence of elements can be encoded into a multiset M of pairs $[i, x]$ where i is the index of the value x. With this encoding, the reaction

$$sort\_bag = \textbf{replace } [i,x],[j,y] \textbf{ by } [i,y],[j,x] \textbf{ if } (j = i+1) \wedge (x > y)$$

replaces a couple of consecutive out-of-order pairs by the couple of consecutive ordered ones. These replacements go on until a stable state is reached, that is to say, when no orderless couple remains. Thus, the final stable solution corresponds to the sorting of the sequence encoded in M.

A more straightforward approach is desirable and possible. A multiset of values in V can be formalized as an element of the free associative and commutative monoid $(V^*, +)$ where $+$ is the operation that merges two multisets. Then, a multiset is a formal sum and a reaction rule is a rewriting rule on a term in $(V^*, +)$ modulo associativity and commutativity [DJ90]. In this framework, the comma between multiset elements in the pattern of the rule[1] is another notation for the $+$ operator.

From this point of view, it is easy to adapt the chemical paradigm to handle sequences: a sequence is an element of a monoid which is only associative. We can use term rewriting modulo associativity to formalize reaction rules on sequences. Thus, reaction:

$$sort\_sequence = \textbf{replace } x, y \textbf{ by } y, x \textbf{ if } x > y$$

applied on a sequence S directly corresponds to a kind of bubble sort. In this rule, the comma in the pattern represents the associative operator of the monoid and is interpreted as the concatenation of sequences. The rule is at the same time more readable because there is no artificial encoding of the sequence data structure into a multiset of pairs, and potentially more efficient because only consecutive elements are matched.

The path followed to extend the chemical paradigm on sequence cannot be easily generalized: rewriting modulo some theory is usually hard and needs *ad hoc* developments. For instance, at this point there is no satisfactory theory

[1] The pattern of the rule is the term between **replace** and **by**.

for rewriting on arrays. However, an alternative framework, focusing on the topological relationships in the data structure, can be developed. This framework encompasses multisets and sequences to include arbitrary data structures.

A Topological Approach. The idea is to consider the comma that appears in the pattern of a rule, not as a *data structure constructor*, but as a *neighborhood relationship* that depends on the data structure on which the rule is applied [GM02a]. In a multiset, all elements are neighbor, which accounts for the associativity and commutativity that enables arbitrary rearrangements of the term that represents the multiset. All other data organizations arise as a restriction of this "universal neighborhood relationship". For instance, in a sequence, the neighborhood relationships are restricted to a linear graph.

By considering various topologies, one may recover well-known computation models: nested multisets correspond to membrane systems [Pău02], constraining the universal topology provided by multisets to nested sequences leads to Lindenmayer systems [RS92]; restriction to discrete lattice corresponds to cellular automata and more general crystalline computations [TM87]. And topologies with higher dimensions can be used to give a direct finite formulation of field equations in classical physical theories [Ton01] with obvious interests for numerical applications [PS93].

2.2 A Short MGS Presentation

The MGS project [GM02b, Gia03] is based on the previous idea: data structures are defined relying on topological notions to specify their neighborhood relationships. In the rest of this section, we show how the notion of data structure can be identified with the notion of field on some underlying space. Such objects can be rewritten, leading to a novel form of case-based definition of function. These notions are illustrated through some simple but informative examples.

Data Structures as Discrete Fields. In MGS, a data structure is handled as a *field* that associates a scalar value to each point of an *underlying space* [GS08]. The structure of this underlying space is of interest for the computation at hand. For example, a multiset of n elements is a field over a space composed of n points. More specifically, the underlying space may represent some meaningful entity for the problem. For instance, in a simulation of the temperature distribution throughout a room, the underlying space is the room. In a distributed computation, the underlying space represents the connectivity between the processing elements.

The spatial structure of the underlying space is used to record the neighborhood relationships needed by the computation. If the computation of the value v associated with a point σ requires the value v' associated with an other point σ', then σ and σ' must be, somehow, neighbors in the underlying space. For example, in a simply linked list, the elements are linearly accessed: the second after the first, the third after the second, etc., inducing an oriented linear space.

In the context of a programming language, the topology of the underlying space must be algebraically defined to avoid the handling of untractable continuous objects. For technical reasons, it is more convenient and more general to associate values with some subspaces of the underlying space rather than with points only (a point being an elementary subspace). Moreover, in this paper we are only interested in the topological properties of the underlying space: the properties related to a metric will not be considered here.

These constraints can be satisfied using *abstract cellular complexes* to specify the underlying space. Abstract cellular complexes are a variant of cellular complexes developed in algebraic topology [Hen94]: a particular class of topological spaces that are partitioned into pieces of elementary space called *topological cells*. Each cell is homeomorphic to an open ball in $\mathbb{R}^d$. By the term *abstract*, we mean here that only the combinatorial structure of cellular complexes is preserved while the geometric characteristic functions mapping cells to open balls are left apart [Kov01].

The corresponding notion of data structure is called *topological collection* in MGS. Topological collections are formalized by *topological chains*, a notion developed in homology theory [Mun84]. Chains are functions that associate values with the cells of abstract cellular complexes. In the following, we will often drop the term "abstract": we only consider abstract cellular complexes and abstract topological cells.

Transformations of Topological Collection. In the chemical paradigm, multiset transformations are defined using rules and can be formalized by associative-commutative rewriting [DJ90]. This schema can be extended to topological collections, relying on the notion of *chain rewriting* defined in [GS08].

The *global transformation* of a topological collection C consists in the parallel application of a set of *local transformations*. A local transformation is specified by a rewriting rule that specifies the changes of a subcollection. An MGS transformation T is accordingly specified by a set of rules:

$$\textbf{trans}\ T = \{\ \ldots;\quad \mathit{rule};\quad \ldots\ \}$$

A rule is a basic transformation that takes the following form:

$$\mathit{pattern}\ \Rightarrow\ \mathit{expression}$$

where *pattern* in the left hand side (lhs) of the rule matches a subcollection B of the collection A on which the transformation is applied. The subcollection B is substituted in A by the collection C computed by the evaluation of the right hand side (rhs) *expression* of the rule.

The Pattern Language. Several pattern languages have been developed in MGS. In this paper, we only consider a subset of the *path patterns*. The grammar of this fragment of pattern expressions is:

$$\beta ::= \ x\ \ |\ \ p, p'\ \ |\ \ p*\ \ |\ \ p\ \textbf{as}\ x\ \ |\ \ p/exp$$

where p, p' are patterns, x ranges over the pattern variables and exp is an expression evaluating to a boolean value. Such patterns can be used to match a *path*: a finite sequence of elements e_i where e_{i+1} is a neighbor of e_i. The explanations below give an informal semantics for these patterns.

variable: a pattern variable x matches exactly one element of the topological collection, that is, a cell σ and its associated value v. A variable can only be defined once in a pattern (patterns are linear) but it may be used elsewhere in the expressions of the rule where it denotes the value v.

neighbor: p, p' is a pattern that matches two connected paths p and p'. The connection relationship depends on the topology of the collection. For example, x, y matches two elements such that y is a neighbor of x.

repetition: pattern $p*$ matches a subcollection of connected elements matched by p.

binding: a binding p **as** x gives the name x to the path matched by p. This name can be used anywhere in the rest of the rule. E.g., the pattern (x, y) **as** d matches two connected elements and the corresponding sequence of two elements can be referred through the variable d.

guard: p/exp matches the collections matched by p such that exp holds. For instance, $y \,/\, y > 3$ matches an element y whose associated value is greater than 3.

The Right Hand Side of a Rule. In a rewriting rule, the lhs and the rhs of the rule denote objects of the same type. For instance, in multiset rewriting, the lhs matches a multiset which is replaced by the multiset computed in the rhs. In term rewriting, the lhs matches a tree which is replaced by the tree computed by the rhs. In graph rewriting, the lhs of a rule matches a graph and the rhs computes a graph to be inserted in place of the matched one. Etc.

Sophisticated data structure make harder the definition of the the rhs and of the associated replacement operation. However, the structure of a path pattern can be used to drastically simplify the rhs of an MGS rule. A *path pattern* matches a path, that is, a sequence. Therefore, the rhs of the rule can evaluate to a sequence, no matter how complex the considered data structure is. The substitution of the matched path by the sequence computed in the rhs is done element-wise. Having a matched path and a rhs sequence of the same length is a constraint that can be relaxed for some collection types. For example, if a transformation is applied on a monoidal collection (i.e. a set, a multiset or a sequence), the rhs can be of arbitrary length. From now on, the expression in the rhs of a rule must be interpreted as a sequence construction.

2.3 The MGS Programming Language

MGS is an experimental programming language that embeds the idea of topological collections and their transformations into the framework of a functional language. Collections are just new kinds of values and transformations are functions acting on collections and defined by a specific syntax using rules. The set

of values has a rich type structure used in the definition of pattern-matching, rule and transformations. The collection types in MGS range from totally unstructured with sets and multisets to more structured with sequences, trees, Voronoï diagrams, Cayley graphs, arbitrary graphs, Generalized Maps [Lie94]... and abstract cellular complexes which subsume all other collection types.

A transformation T is a function on collections and a *first-class* value. For instance, a transformation can be passed as an argument to another function or be returned as a result. This feature allows to sequence and compose transformations very easily.

The expression $T(c)$ denotes the application of one transformation step to the collection c. As said above, a transformation step consists in the application of the rules (modulo the rule application's strategy). A transformation step can be easily iterated:

$$T[n]\,(c) \quad \text{denotes } n \text{ iterations of the application of } T \text{ on } c$$
$$T[\textbf{fixpoint}]\,(c) \quad \text{denotes the application of } T \text{ until a fixpoint is reached}$$

Several rule application strategies and transformation application strategies have been defined, including asynchronous and stochastic ones [SMC+08]. Synchronous rule application strategies (several rules are applied in parallel in one application of a transformation) are non-deterministic: only non-intersecting paths are rewritten and these paths are non-deterministically chosen (priorities or probabilities can be used to have a finer control). For example, the *maximal parallel application strategy* apply as many rules as possible in parallel that is, when rewriting occurs, there is no path in the remaining elements that can be matched by the lhs of a rule. This strategy is similar to the one used in Lindenmayer systems [RS92].

2.4 A Few Examples

Obviously, all Gamma chemical programs can be translated easily in MGS. We give below some examples that take advantage of the spatial structure of the topological collection. After two simple examples on sequences, we focus on computations on a regular grid and on a graph because such data structures are not algebraic data structures, and therefore the usual approach based on term rewriting is not applicable.

MGS has been involved in sophisticated simulation applications in biology, like neurulation [SM07], cell mobility [SM05], growth of plant meristem at a cellular level [BdR+06], or simulations at various scales of a synthetic multicellular bacterium [IGE07]. Classical algorithms (various sorting procedures, graph algorithmics, optimization processes, mesh refinement algorithms, etc.) have also been easily developed, see [Mic07].

Bubble Sort. In the MGS syntax, the *sort_sequence* program in section 2.1 is specified by the following transformation:

$$\textbf{trans}\ \mathit{sort\_sequence} = \{\ x, y\ /\ x{>}y\ \Rightarrow\ y, x\ \}$$

As mentioned above, the rhs of the rule is a sequence construction: the comma is then interpreted as the sequence constructor. The transformation must be iterated until a fixed point is reached. Note that the fixed point is reached regardless the rule application strategy. This result can be established by the tools presented in section 3.

Duplicate Removal. It is easy to remove the contiguous duplicated elements in a list using the iteration of the transformation:

$$\textbf{trans}\ remove\_duplicate = \{\ x, y\ /\ x{=}y\ \Rightarrow\ x\ \}$$

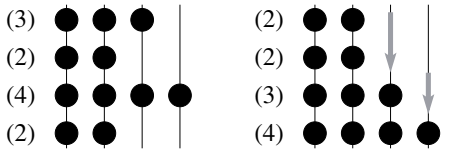
The pattern $x, y\ /\ x{=}y$ selects two contiguous elements labeled by the same value. Such occurrences are replaced by only one element. Note that the rhs must be interpreted as a sequence of only one element. The conversion between an element and the corresponding singleton is implicit.

Bead Sort. Various meshes can be described in MGS. For example, the NEWS lattice is specified by the following type declaration:

$$\textbf{gbf}\ NEWS = \langle North, East, West, South; North + South = 0, East + West = 0\rangle$$

This declaration introduces a "group based data-field" or GBF [GM01]. The underlying space of a GBF is the Cayley graph of the abelian group presentation specified by the right hand side of the declaration. The vertices of the Cayley graph are linked by edges labeled by the group generators *North*, *East*, etc.

The bead-sort is an original way to sort positive integers proposed by [ACD02]. This sorting algorithm considers a column of numbers written in unary basis. The first schema below pictures the numbers 3, 2, 4 and 2 where the beads stand for the digits. The sorting is done by letting the beads fall down as shown on the second schema. The numbers can be implemented by a regular grid of booleans where *true* stands for a digit and *false* for the absence of digit as shown on the third and fourth schema.



The bead-sort is achieved by iterating the application of the following transformation until a fixpoint is reached:

$$\textbf{trans}\ bead\_sort = \{x/(x = false)\quad |North{>}\quad y/(y = true)\ \Rightarrow\ y, x\}$$

The construction $|North{>}$ refines the comma operator, constraining the element y to be a *North*-neighbor of x.

Hamiltonian Path. A graph is an MGS topological collection. Expressing in MGS the search of an Hamiltonian path in a graph is straightforward:

exception *Not_Found*, *Found*;;
trans $H = \{\ x* \ \textbf{as}\ path/size(path) = size(\textbf{self}) \ \Rightarrow\ \textbf{raise}\ Found(path)\ \}$;;
fun $hamiltonian(g) =$ **try**
$H(g)$; **raise** *Not_Found*
with $Found(path) \rightarrow path$;;

Transformation H uses an iterated pattern $x*$ that matches a path. The keyword **self** refers to the collection the transformation is applied on. The size of a graph returns the number of its vertices. So, if the length of *path* is the same as the number of vertices in the graph, then *path* is an Hamiltonian path (patterns are linear without repeated matched element). The rhs raises an exception which is trapped in function *hamiltonian*. The normal return of H is followed by the raising of the *Not_Found* exception in function *hamiltonian*.

3 From Local Changes to Global Specifications

The notions introduced in MGS for spatial organization can be used to handle directly: (a) highly organized data structures used in algorithms (like trees, arrays, etc.), (b) semi-structured data like those managed in XML applications (XML schema are well represented by nested topological collections) or (c) to take into account the data distribution over a network. The network architecture can be static and regular (e.g., as in a tightly coupled parallel architecture) or more fuzzy and dynamic (e.g., as in a grid on the Internet, in a P2P system or in an amorphous medium [AAC+00]). In either case, the communication cost between processing elements induces a neighborhood relationship. The states of the processing elements together with this neighborhood relationship constitute a topological collection. If any point-to-point communication exhibits the same uniform cost, the corresponding topology is the topology of a multiset.

3.1 Self-∗ Properties and Fixed Point Iterations of Local Rules

In this point of view, the state of an autonomic system is a (distributed) topological collection. The evolution of an autonomic system is specified through local evolution rules that define the (local) evolution of a small subpart of the system. A topological collection stable w.r.t. the reactions represents an admissible state of the autonomic system. A perturbation or an interaction with the environment corresponds to a change in the topological collection: the addition or the removal of some elements *or* a modification of the neighborhood relationship. This framework is illustrated in Fig. 1.

Thus, an autonomic system can be defined in this framework if one can infer that the asynchronous parallel applications of local rules lead to stable points exhibiting some required properties. Some theoretical tools can be used in this

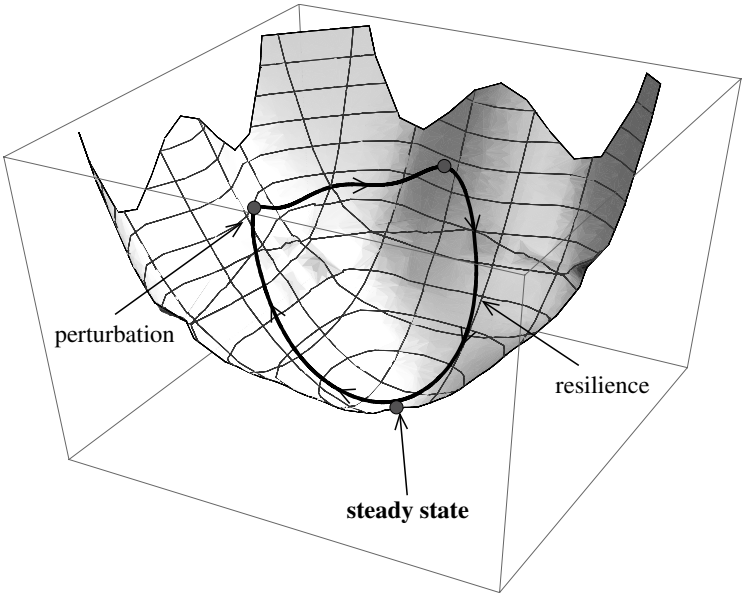


Fig. 1. Autonomic computing via trajectory stabilization. An autonomic system can be seen as a distributed dynamical system. In this diagram, the states of the dynamical system are figured as a plane and the system's evolutions are given by a trajectory. The surface represents some potential, for instance a quantitative evaluation of the divergence of the system from a desired behavior. When some transient perturbations make the system leave its steady state, the local transformations triggered by the matching of some rules eventually lead to the return of the system's state to an admissible state.

difficult task. In the rest of this presentation, we develop several topological collection orderings based on *multiset orderings* [DM79]. Such orderings can be used to prove the convergence of an autonomic system towards a fixed point.

The framework pictured in Fig. 1 can be made more precise in the following way. We suppose that the state of an autonomic system is described by a topological collection $c \in \mathcal{C}$ where $\mathcal{C}$ is the state space of the system. The autonomic program driving the autonomic system is specified by a unique transformation T. The admissible states s of the corresponding autonomic system are fixed points of T: $s = T(s)$. A perturbation of the system corresponds to a state s' such that $s' \neq T(s')$. The trajectory of the system after the perturbation is given by the sequence $s_0 = s', s_1 = T(s_0), \ldots, s_{n+1} = T(s_n)$. The problem is then to know whether the sequence s_n converges towards a fixed point in a finite number of steps. A classical approach is to exhibit a well-founded ordering [2] $\prec$ of the state space $\mathcal{C}$ such that $s_{n+1} \prec s_n$.

[2] An ordering is well-founded if it contains no infinite descending sequences of elements. For countable sets, a well-founded ordering $\prec$ can be specified by giving a mapping τ onto the positive integers such that $c \prec c' \Leftrightarrow \tau(c) < \tau(c')$. The "potential" surface in Fig. 1 figures such a function τ.

In this context, it would be very useful to exhibit various well-founded orderings on arbitrary topological collections. To this end, we can adapt the well-known multiset ordering introduced in [DM79].

3.2 Multiset Ordering

For a partially-ordered set of values $(V, <)$, the multiset ordering $\ll$ on $\mathcal{M}(V)$ the multisets over V, is defined as follows: $A \ll B$ if for some $X, Y \in \mathcal{M}(V)$ such that $X \neq \emptyset$ and $X \subset B$,

$$A = (B - X) \cup Y \qquad \text{and} \qquad \forall y \in Y, \exists x \in X, y < x$$

In other words, A is obtained from B by removing some elements (those in X) and their replacement with a finite number of elements (those in Y) that are smaller than one of the element of X. The definition of the relation $\ll$ relies on the relation $<$ and we will write $\ll_<$ when we need to make such dependence explicit.

In the previous definition, set operators denote their multiset analogs: the equality $A = B$ of two multisets, for example, means that any element occurring exactly n times in A, also occurs exactly n times in B. The union of two multisets $A + B$ is a multiset containing $m + n$ occurrences of any element occurring m times in A and n times in B. $A \subset B$ means that for any element occurring n times in A, this element occurs m times in B with $n < m$. If $A \subset B$, then $B - A$ is a multiset where any element occurring n times in A and m times in B, occurs $m - n$ times. Union and difference between multisets are extended to the addition and the removal of elements: $A + x = A + \{x\}$ and $A - x = A - \{x\}$.

The result demonstrated in [DM79] is that $\ll_<$ is well-founded iff $<$ is well-founded. This result can be generalized in two ways to topological collections more general than multisets.

3.3 Forgetting the Spatial Structure

The first approach simply consists in forgetting the additional spatial structure of a topological collection. Thus, we can associate to each topological collection c the multiset $\mathbf{m}(c)$ of the values associated with the cells of c. Assuming that $\ll$ is well-founded, the order $\prec$ defined by $c \prec c'$ iff $\mathbf{m}(c) \ll \mathbf{m}(c')$, is also well-founded. We write $\prec_\ll$ when we want to make explicit the underlying multiset ordering.

A Toy Example. To illustrate the use of a well-founded topological collection ordering $\prec$, we consider a wireless sensor network targeted at environmental monitoring, like for example the one described in [SKHH06]. Each node reacts to environmental changes and, to fix the idea, the goal of the system is to record the maximal temperature over the covered area. For this purpose, each node i stores a current maximal temperature t_i and updates it by comparison with its neighbors. A perturbation is the change of one t_i to record a new local maximum.

The state of the autonomic system is a topological collection $c \in \mathcal{C}$ where $\mathcal{C}$ can be for instance the set of vertex-labeled undirected graphs (a vertex represents

a node, an edge between two vertices corresponds to nodes able to interact, and the label of the vertex i is the current maximal temperature t_i). We further suppose that there is only one strongly connected component. The behavior of the system is specified as the iteration of the following transformation:

$$\textbf{trans}\ \mathit{propagate} = \{\ t, t'/t' < t\ \Rightarrow\ t, t\ \}$$

(the local change of a t_i is considered as an external change and not modeled here).

It is straightforward to prove that in absence of a perturbation, the system will reach a state where all t_i are equal. This property is a global one and must be deduced from the application of the local rule. Although this result is obvious, its proof illustrates the use of a well-founded topological collection ordering.

Constant Fields are the Fixed Points. First, the topological collections c where each cell has the same value t, are the only fixed points of the *propagate* transformation. Indeed, if c is a fixed point, we cannot find in c a pair of cells σ and σ' labeled by the values t and t' such that $t < t'$ (the relation $<$ is the classical numerical comparison over the integers).

Exhibiting a Well-Founded Ordering. We may suppose that the temperature is bounded by a sufficiently big positive number $t_{\max}$. Thus the set $V = \{-\infty, \ldots, t_{\max}\}$ with the order $\lessdot$ defined by $(t \lessdot t') \Leftrightarrow (t' < t)$ is well-founded. Consequently, the order on $\mathcal{M}(V)$ defined by $\ll_{\lessdot}$ is well-founded, as well as $\prec_{\ll_{\lessdot}}$ on $\mathcal{C}$.

The System's Trajectory is Decreasing. As a matter of fact, if $c' = \mathit{propagate}(c)$ and $c' \neq c$, we have $c' \prec_{\ll_{\lessdot}} c$ because $\mathbf{m}(c') = \mathbf{m}(c) - t + t'$ with $t, t' \in \mathbf{m}(c)$ and $t' \lessdot t$, and so $\mathbf{m}(c') \ll_{\lessdot} \mathbf{m}(c)$. This just shows that the sequence $T^n(c)$ is decreasing and since the order is well-founded, it implies that it converges in a finite number of steps to some fixed point.

3.4 Topological Collection as Generalized Multisets

Forgetting the spatial structure is useful only if the fixed point does not depend on the spatial structure of the topological collection, which is generally not true.

It is possible at the same time to keep the spatial structure of a topological collection c and to consider it as a kind of generalized multiset. To see the connection between both notions, we need to introduce some formal definitions. The reader is warned that the following definitions are truncated to only focus on the multiset structure of a topological collection. The algebraic structure required to represent the spatial organization of a topological collection (dimension of a cell, boundary operator, the neighborhood relationships, etc.) is ignored. Complete definitions can be found in [GM02b, GS08].

Definition 1 (Cells, Abstract Cellular Complexes and Chains). *Let* S *be a set of symbols called the* universal set of cells*. An abstract cellular complex* K *is a partially ordered subset of* S*. We write* $\mathbb{K}$ *the set of all abstract cellular complexes.*

Let K *be a complex and* G *be an abelian group. The set of functions from* K *to* G*, null almost everywhere, is called the set of* topological chains *of* K *to* G*, and is written* $\mathcal{C}_\mathsf{K}(G)$*.*

The elements of $\mathcal{C}_\mathsf{K}(G)$ are easily representable by finite formal sums:

$$\forall c \in \mathcal{C}_\mathsf{K}(G), \quad c = \sum_{\sigma \in \mathsf{K}} c(\sigma).\sigma$$

where $c(\sigma).\sigma$ is the formal product that represents the association of the value $c(\sigma) \in G$ with the cell $\sigma \in \mathsf{S}$. Thanks to the abelian group structure imposed on G, the set $\mathcal{C}_\mathsf{K}(G)$ is an abelian group considering the addition $+_{\mathcal{C}_\mathsf{K}(G)}$ defined by:

$$\forall c_1, c_2 \in \mathcal{C}_\mathsf{K}(G), \quad c_1 +_{\mathcal{C}_\mathsf{K}(G)} c_2 = \sum_{\sigma \in \mathsf{K}} (c_1(\sigma) +_G c_2(\sigma)).\sigma$$

where $+_G$ denotes the group operation in the group G. The proof is straightforward. If the context is clear, the subscript in the notation of the group operation will be dropped.

While they seem useless in our context, the group structure on the set of values and the induced group structure on chains are really meaningful to deal with topological collections:

- $0_G.\sigma$ means that no value is associated with σ (0_G is the neutral element in the group G),
- the neutral element of $\mathcal{C}_\mathsf{K}(G)$ represents the empty data structure,
- the operator $+_{\mathcal{C}_\mathsf{K}(G)}$ adds a new association of a value $v \in G$ with a cell $\sigma \in \mathsf{K}$ in a data structure c: $c + v.\sigma$,
- the opposite $-_{\mathcal{C}_\mathsf{K}(G)}$ removes an association: $c - v.\sigma = c + (-v).\sigma$.

We have mentioned above that a topological collection c can be represented as a finite formal sum $\sum_{\sigma \in \mathsf{K}} c(\sigma).\sigma$. We interpret this sum as a set of pairs

$$\mathbf{mp}(c) = \{(\sigma, c(\sigma)) : \sigma \in \mathsf{S} \text{ and } c(\sigma) \neq 0_V\}$$

A set is a special kind of multiset and so can be ordered using a multiset ordering based on the ordering of the set elements. Hence the desired definition and theorem:

Theorem 1. *Let* $(V, <)$ *a partially-ordered set of values, and* $(\mathsf{S}, \lessdot)$ *a partially-ordered universal set of cells. Then the* topological collection ordering *is the partial-order* $\prec$ *defined on* $\mathcal{C}_\mathsf{K}(V)$ *by:* $c \prec c'$ *iff* $\mathbf{mp}(c) \prec_{(\lessdot \times <)} \mathbf{mp}(c')$ *where* $(\lessdot \times <)$ *is the lexicographic or the element-wise ordering on* $\mathsf{S} \times V$*. The relation* $\prec$ *is well-founded iff* $\lessdot$ *and* $<$ *are well-founded.*

In words, a collection c is smaller than a collection c' if c can be obtained from c' by replacing some valued cells by some other valued cells, in arbitrary number, provided that the introduced cells and/or the associated values are smaller.

3.5 Correction of the Eratosthenes's Sieve

To illustrate the previous result, we consider the fixed point iteration of the following transformation[3]:

$$
\begin{aligned}
&\textbf{trans}\ \mathit{prime} = \{\\
&\quad x/(x>0),\ 0 \ \Rightarrow\ 0, -x;\\
&\quad x/(x\geq 0),\ y/(y<0),\ 1/(-y>x) \ \Rightarrow\ x, -y, 1;\\
&\quad x/(x\geq 0),\ y/(y<0),\ 1/(-y<x) \ \Rightarrow\ x, 1;\\
&\quad x/(x\geq 0),\ y/(y<0),\ z/(z>0)/(-y>x)\wedge(-y<z) \ \Rightarrow\ x, -y, y, z;\\
&\quad x/(x<0),\ y/(y>0)\wedge(y\leq -x)\wedge(x\%y\neq 0) \ \Rightarrow\ y, x;\\
&\quad x/(x<0),\ y/(y>0)\wedge(y\leq -x)\wedge(x\%y= 0) \ \Rightarrow\ y;\\
&\quad x/(x<0),\ y/(y>0)\wedge(y> -x)\wedge(y\%x\neq 0) \ \Rightarrow\ y, x;\\
&\quad x/(x<0),\ y/(y>0)\wedge(y> -x)\wedge(y\%x= 0) \ \Rightarrow\ x;\\
&\}
\end{aligned}
$$

We assume that this process is applied on a sequence that begins with a zero and ends with a one. This transformation maintains a sorted sequence (sorted at the exception of the ending one) of relatively prime positive integers. An external perturbation consists in introducing an arbitrary integer $x > 0$ at the beginning of the sequence. If all the integers up to $m > 1$ are introduced, *in any order*, the sequence stabilizes on the sorted sequence of prime numbers up to m.

In the rules, the operator $\wedge$ is the boolean conjunction operator and $\%$ is the modulo operator. The fourth rule increases the length of the sequence while some other rules shrink it. So the convergence is not obvious. The idea behind this program is similar to the sieve of Eratosthenes but adapted in order to admit the orderless introduction of the integers. The perturbation x "travels" along the sequence, from the beginning to the end of the sequence. This "traveling number" is distinguished from the other numbers in the sequence using a negative number (note that there is no constraint in the perturbation, so several "traveling numbers" may coexist in the sequence). This number and the next are tested to check if they are relatively prime. The "traveling number" is inserted in the sequence at the correct position in order to maintain a sorted sequence and the propagation continues to check the primality of the rest of the sequence.

The correction of this algorithm is easy to establish with the previous tool. We focus on the convergence, that is: a fixed point is reached in a finite number of steps. It is sufficient to exhibit a well-founded order such that a rule application to sequence c_1 gives raise to a smaller sequence c_2. We take the topological collection order induced by the following order on cells and values:

- Cells are ordered in descending order "from left to right". For instance, the sequence $0, 2, 3, 1$ is represented by the sum $0.\sigma_0 + 2.\sigma_1 + 3.\sigma_2 + 1.\sigma_3$ and

[3] This transformation is derived from a natural implementation of the sieve of Eratosthenes, see [GM02b]. This program is not intended to be readable and has been chosen to illustrate the approach on an arbitrary set of rules and conditions.

$\sigma_3 \lessdot \sigma_2 \lessdot \sigma_1 \lessdot \sigma_0$. We assume that only a finite number of cells has been used, so $\lessdot$ is well-founded. This assumption is satisfied because a cell is created each time a new integer is introduced and we suppose that there is a bounded number of introductions.
- Relative integers are ordered as follows: let N be the greatest integer introduced in the sequence. Then the set of values $V = \{-N, \ldots, N\}$ used in the program is ordered by : $-N > -N+1 > \cdots > -1 > N > N-1 > \cdots > 1 > 0$. This order is well-founded. Note that V is a subset of the abelian group $(\mathbb{Z}, +)$ and this is enough for our purpose.

Finally, we consider the lexicographic order on the pairs (cell, value). It is straightforward to check that each rule applied to a sequence c_1 gives raise to a smaller sequence c_2. We will sketch only three rules for illustration:

- The first rule does not change the sequence length. Its application replaces in a sequence c_1 the submultiset $x.\sigma_0 + 0.\sigma_1$ by $0.\sigma_0 + (-x).\sigma_1$ where $\sigma_1 \lessdot \sigma_0$ and $x > 0$. We have $x.\sigma_0 > 0.\sigma_0$ (because $x > 0$) and $x.\sigma_0 > (-x).\sigma_1$ (because $\sigma_1 \lessdot \sigma_0$). It follows that $c_2 \prec c_1$.
- Rule 4 is the only rule that increases the sequence length. The submultiset:

$$c_1 = x.\sigma_0 + y.\sigma_1 + z.\sigma_2 \qquad \text{where } \sigma_2 \lessdot \sigma_1 \lessdot \sigma_0 \text{ and } 0 \leq x \text{ and } y < 0$$

  is replaced by

$$c_2 = x.\sigma_0 + (-y).\sigma_1 + y.\sigma_2 + z.\sigma_2' \qquad \text{where } \sigma_2' \lessdot \sigma_2$$

  We have $c_2 \prec c_1$ because $y.\sigma_1 > (-y).\sigma_1$ (y is strictly negative and so $-y > y > 0$). We have also $y.\sigma_1 > y.\sigma_2$ and $y.\sigma_1 > z.\sigma_2'$ because $\sigma_2' \lessdot \sigma_2 \lessdot \sigma_1$.
- The application of rule 8, the last rule of the transformation, decreases the sequence since it removes one element.

So, since the successive sequences are decreasing in a well-founded order the sequence must converge in a finite number of steps.

3.6 The Group Structure and the Ordering of the Values

We cannot assume that every set of values V we may consider, carries a group structure. The group operation is useful to manage the association of a value to a cell but it is in some sense external to the proper definition of V. Hopefully, in absence of any "natural" group operation on V, we can use a mathematical trick to turn any set V into an abelian group.

Definition 2 (Abelianization of a set). *Let V be an arbitrary set. The $\mathbb{Z}$-module freely generated by the elements of V, written $\mathcal{A}(V)$, is the free abelian group generated by the elements of V. An element g of $\mathcal{A}(V)$ can be written has a formal sum: $\sum_{v \in V} z_v.v$ where z_v belongs to $\mathbb{Z}$.*

In the following, we consider only "finite formal sum" where only a finite subset of coefficients z_v are different from zero. The structure of $\mathbb{Z}$-module generalizes the structure of multiset: as a matter of fact, any multiset $m \in \mathcal{M}(V)$ can be represented by a formal sum with positive coefficients z_v. So, $\mathcal{A}(V)$ generalizes $\mathcal{M}(V)$ by allowing a *negative* number of occurrences.

Given $g \in \mathcal{A}(V)$, we can distinguish between the positive and the negative coefficients of g, that is:

$$g = (\sum_{v \in V_1} z_v.v) - (\sum_{v' \in V_2} z_{v'}.v)$$

with $V_1, V_2 \subset V$. Assuming $V_1 \cap V_2 = \emptyset$ and all coefficients $z\_$ strictly positive, the decomposition of g is unique and we write: $g = g^+ - g^-$. Both sums g^+ and g^- are multisets on V and this justify the following definition of the abelian ordering.

Definition 3 (Abelian ordering). *Let $(V, <)$ be a partially-ordered set of values. The* abelian ordering *$\ll$ on $\mathcal{A}(V)$ is defined as follows: $g \ll g'$ iff $(g^+, g^-) < (g'^+, g'^-)$ where $<$ is the lexicographic ordering or the element-wise ordering and where the elements of the pair are compared using the multiset ordering on $\mathcal{M}(V)$.*

Theorem 2. *Let $(V, <)$ a well-founded partially-ordered set of values, and $(\mathsf{S}, \lessdot)$ a well-founded partially-ordered universal set of cells. Let $\prec$ be the* topological collection ordering *defined on $\mathcal{C}_\mathsf{K}(Abel(V))$ by: $c \prec c'$ iff $\mathbf{mp}(c) \prec_{(\lessdot \times \ll)} \mathbf{mp}(c')$ where $\ll$ is the abelian ordering induced by $<$ on $\mathcal{A}(V)$ and $(\lessdot \times \ll)$ is the lexicographic or the element-wise ordering on $\mathsf{S} \times \mathcal{A}(V)$. Then, the relation $\prec$ is well-founded.*

4 Conclusion

In this paper we have shown how the declarative chemical programming paradigm can be enhanced to take into account logical and physical spatial organization using some notions developed in algebraic topology. This new framework is investigated in the MGS experimental programming language. At the core of this extension is the idea that a data structure can be conceived as a physical field. This notion is not entirely new and we review below some previous work.

The chemical paradigm has been advocated for the development of amorphous and autonomic systems. In a second part, we have adapted the well-founded multiset ordering, a classical tool used to prove program termination, to show how the convergence of a fixed point iteration can be established for topological collections. The parallel between autonomic systems and self-stabilizing systems has been recently noticed and we sketch some differences.

Data Structure as Field. The notion of data field is an old one in computer science: it already appeared in the development of recurrence equations and

dates at least from [KMW67]. The term "data field" seems to be used for the first time in [CiCL91]. The notions of data field and data parallelism have been explicitly brought together in [Lis93]. This approach is also close to the notion of *pvar* or *xapping* [SH86] in the context of the *Connection Machine*. However, in all these works, the set of points is simply an integer lattice (points are elements of $\mathbb{Z}^n$) and is often left implicit.

Topological collections consider, for the underlying space, more general spaces than integer lattices or even arbitrary graphs, in order to accommodate a large variety of spatial organizations [GM02a]. This generality will ease the development of various applications, for example in simulation by allowing a direct representation of the modeled entities. As a matter of fact, many physical quantities have different values at various points in space (temperature field, velocity field, potential, etc.). In addition, the value associated with a spatial domain often depends on the dimension of the domain [Ton74], e.g. a concentration for a volume and a flux for the surface bounding this volume. Such generality will also facilitate portability by offering a uniform abstraction of arbitrary spatial computing media (e.g.: grids, amorphous computers, chemical reaction diffusion computers, DNA self-assembly, natural or synthetic cellular assemblies, etc.).

Self-Stabilization and the Self-∗ Paradigms. Recent works in the self-stabilization community [BDHY07, BDH+08] advocate the use of self-stabilization as a provable property to achieve the goal of self-∗ paradigms for systems. Usually, a self-stabilizing system is designed to start in any possible configuration where processors, processes, communication devices, etc. are in an arbitrary state. The approach exemplified here follows the same line. However, it is more abstract (communications are abstracted by the data movements within a topological collection) and less demanding concerning the initial state. We want to underline that our goal is not to develop algorithmic tools for designing self-stabilizing systems but to show that well-known approaches in *program semantics* can be adapted to new programming paradigms advocated for amorphous and autonomic applications.

Future Work. Our future research directions follow two paths. First, we need to investigate further how classical tools in the semantics of programs can be adapted to the case of perpetual autonomic systems. The other direction tries to import some tools from dynamical system theory to design and study the semantics of autonomic systems. A first step in this direction is the development of a discrete analog of differential operators for topological collections [GS08]. Our idea is to rely on topological and geometrical results (fixed-point theorem, existence of objects defined by differential equations, integration theorem) to design, control and validate global behaviors from the specification of local ones.

Acknowledgments. Pascal Fradet at INRIALPES, Thierry Priol and Jean-Pierre Banâtre at IRISA are gratefully acknowledged for stimulating discussions and interactions on the chemical paradigm an its application to autonomic systems. The authors also wish to thank the organizers of the InterLink workshop series

for making these fertile workshops possible. The work presented here are partially funded by the ANR NanoProg, the ANR AutoChem, a BQR of the University of Evry and the CNRS.

References

[AAC+00] Abelson, H., Allen, D., Coore, D., Hanson, C., Homsy, G., Knight, T.F., Nagpal, R., Rauch, E., Sussman, G.J., Weiss, R.: Amorphous computing. CACM: Communications of the ACM 43 (2000)

[ACD02] Arulanandham, J.J., Calude, C.S., Dinneen, M.J.: Bead-Sort: A natural sorting algorithm. EATCS Bull. 76, 153–162 (2002)

[BCM88] Banâtre, J.-P., Coutant, A., Le Metayer, D.: A parallel machine for multiset transformation and its programming style. Future Generation Computer Systems 4, 133–144 (1988)

[BDH+08] Brukman, O., Dolev, S., Haviv, Y., Lahiani, L., Kat, R., Schiller, E., Tzachar, N., Yagel, R.: Self-stabilization from theory to practice. Bulletin of the EATCS (94), 130–150 (2008)

[BDHY07] Brukman, O., Dolev, S., Haviv, Y., Yagel, R.: Self-stabilization as a foundation for autonomic computing. In: Proc. of the Second IEEE International Conference on Availability, Reliability and Security (ARES 2007), Workshop on Foundation of Fault-tolerance Distributed Computing (FOFDC 2007), pp. 991–998. IEEE, Los Alamitos (2007)

[BdR+06] Barbier de Reuille, P., Bohn-Courseau, I., Ljung, K., Morin, H., Carraro, N., Godin, C., Traas, J.: Computer simulations reveal properties of the cell-cell signaling network at the shoot apex in Arabidopsis. PNAS 103(5), 1627–1632 (2006)

[BL90] Banâtre, J.-P., Le Métayer, D.: The GAMMA model and its discipline of programming. Science of Computer Programming 15(1), 55–77 (1990)

[BRF04] Banâtre, J.-P., Radenac, Y., Fradet, P.: Chemical specification of autonomic systems. In: IASSE, pp. 72–79. ISCA (2004)

[CiCL91] Chen, M., Il Choo, Y., Li, J.: Crystal: Theory and Pragmatics of Generating Efficient Parallel Code. In: Szymanski, B.K. (ed.) Parallel Functional Languages and Compilers. Frontier Series, vol. 7, pp. 255–308. ACM Press, New York (1991)

[DJ90] Dershowitz, N., Jouannaud, J.-P.: Rewrite systems. In: Handbook of Theoretical Computer Science, vol. B, pp. 244–320. Elsevier Science, Amsterdam (1990)

[DM79] Dershowitz, N., Manna, Z.: Proving termination with multiset orderings. Communications of the Association for Computing Machinery 22, 465–476 (1979)

[Gia03] Giavitto, J.-L.: Invited talk: Topological collections, transformations and their application to the modeling and the simulation of dynamical systems. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, Springer, Heidelberg (2003)

[GM01] Giavitto, J.-L., Michel, O.: Declarative definition of group indexed data structures and approximation of their domains. In: Proceedings of the 3nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001). ACM Press, New York (2001)

[GM02a] Giavitto, J.-L., Michel, O.: Data structure as topological spaces. In: Calude, C.S., Dinneen, M.J., Peper, F. (eds.) UMC 2002. LNCS, vol. 2509, pp. 137–150. Springer, Heidelberg (2002)

[GM02b] Giavitto, J.-L., Michel, O.: The topological structures of membrane computing. Fundamenta Informaticae 49, 107–129 (2002)

[GS08] Giavitto, J.-L., Spicher, A.: Topological rewriting and the geometrization of programming. Physica D (2008) (accepted for publication)

[Hen94] Henle, M.: A combinatorial introduction to topology. Dover publications, New-York (1994)

[Hor01] Horn, P.: Autonomic computing: IBM's perspective on the state of information technology. Technical report, IBM Research (October 2001), http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf

[IGE07] IGEM. Modeling a synthetic multicellular bacterium. Modeling page of the Paris team wiki at iGEM 2007 (2007), http://parts.mit.edu/igem07/index.php/Paris/Modeling

[KMW67] Karp, R.M., Miller, R.E., Winograd, S.: The organization of computations for uniform recurrence equations. Journal of the ACM 14(3), 563–590 (1967)

[Kov01] Kovalevsky, V.: Algorithms and data structures for computer topology. In: Digital and image geometry: advanced lectures, pp. 38–58. Springer, New York (2001)

[Lie94] Lienhardt, P.: N-dimensional generalized combinatorial maps and cellular quasi-manifolds. International Journal on Computational Geometry and Applications 4(3), 275–324 (1994)

[Lis93] Lisper, B.: On the relation between functional and data-parallel programming languages. In: Proc. of the 6th. Int. Conf. on Functional Languages and Computer Architectures. ACM Press, New York (1993)

[Mic07] Michel, O.: There's plenty of room for unconventional programming languages, or, declarative simulations of dynamical systems (with a dynamical structure). Habilitation Manuscript (December 2007), http://www.ibisc.univ-evry.fr/~michel/Hdr/hdr.pdf

[Mun84] Munkres, J.: Elements of Algebraic Topology. Addison-Wesley, Reading (1984)

[Pău02] Păun, G.: Membrane Computing. An Introduction. Springer, Berlin (2002)

[PS93] Palmer, R.S., Shapiro, V.: Chain models of physical behavior for engineering analysis and design. In: Research in Engineering Design, vol. 5, pp. 161–184. Springer International, Heidelberg (1993)

[RS92] Rozenberg, G., Salomaa, A. (eds.): Lindenmayer Systems: Impacts on Theoretical Computer Science, Computer Graphics and Developmental Biology. Springer, Heidelberg (1992)

[SH86] Steele, G.L., Hillis, D.: Connection machine LISP: Fine grained parallel symbolic programming. In: Proceedings of the 1986 ACM Conference on LISP and Functional Programming, pp. 279–297. ACM, New York (1986)

[SKHH06] Suhonen, J., Kohvakka, M., Hännikäinen, M., Hämäläinen, T.D.: Design, implementation, and experiments on outdoor deployment of wireless sensor network for environmental monitoring. In: Vassiliadis, S., Wong, S., Hämäläinen, T.D. (eds.) SAMOS 2006. LNCS, vol. 4017, pp. 109–121. Springer, Heidelberg (2006)

[SM05] Spicher, A., Michel, O.: Using rewriting techniques in the simulation of dynamical systems: Application to the modeling of sperm crawling. In: Sunderam, V.S., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2005. LNCS, vol. 3514, pp. 820–827. Springer, Heidelberg (2005)

[SM07] Spicher, A., Michel, O.: Declarative modeling of a neurulation-like process. BioSystems 87(2-3), 281–288 (2007)

[SMC+08] Spicher, A., Michel, O., Cieslak, M., Giavitto, J.-L., Prusinkiewicz, P.: Stochastic p systems and the simulation of biochemical processes with dynamic compartments. BioSystems 91(3), 458–472 (2008)

[TM87] Toffoli, T., Margolus, N.: Cellular automata machines: a new environment for modeling. MIT Press, Cambridge (1987)

[Ton74] Tonti, E.: The algebraic-topological structure of physical theories. In: Glockner, P.G., Sing, M.C. (eds.) Symmetry, similarity and group theoretic methods in mechanics, Calgary, Canada, pp. 441–467 (August 1974)

[Ton01] Tonti, E.: A direct discrete formulation of field laws: The cell method. Computer Modeling in Engineering & Sciences 2(2), 237–258 (2001)

Emerging Models of Computation: Directions in Molecular Computing

Position Paper for InterLink Workshop, May 2007

Darko Stefanovic

Department of Computer Science
University of New Mexico

1 Trends in Mainstream Computing

Computing as we have known it for 60 years is based on the von Neumann stored-program concept and its ubiquitous implementation in the form of electronic instruction processors. For the past four decades, processors have been fabricated using semiconductor integrated circuits, the dominant material being silicon, and the dominant technology CMOS. Relentless miniaturization has been decreasing feature size and increasing both the operating frequency and the number of elements per chip, giving rise to so-called Moore's law (which we interpret broadly to mean the expectation of an exponential improvement in salient performance parameters).

Much has been said about the imminent demise of Moore's law. Yet, hardware engineers keep finding ways to extract more performance out of silicon processes and architects out of instruction processors. New nanometer-scale processes are coming on line, new copper interconnects promise renewed clock speedups. Meanwhile, multiple cores on a chip provide an easy way to redeem chip real estate.

The trend of the past is likely to continue, unchanged in the main, for some time. Note, however, that deployment of multiple cores in each chip, thus in each desktop computer, will demand reconsidering how performance is to be delivered to the individual applications run by the individual user. While certain graphics and video applications offer easily exploited parallelism, many others do not, and a challenge in this domain will be to make them usably parallel without heroic effort on the part of the programmer. While this problem is being addressed on the software side, hardware tweaks permit using spare cores for a boost in sequential performance [1].

In due course, fundamental scaling limits will be encountered, most of all the problem of heat dissipation inherent to devices in which an electronic charge is used for state representation. Completely new alternatives to traditional silicon (CMOS) fabrication are being pursued at the level of devices, such as single-electron transistors, carbon nanotubes, silicon nanowires, molecular switches, nanomagnets, quantum dots, chemically assembled electronics, chemical logic gates with optical outputs, and three-dimensional semiconductor integration.

Alternative architectures (beyond instruction processors) are also being explored, such as amorphous computing [2] , spatial computing [3], blob computing [4], cell

M. Wirsing et al. (Eds.): Software-Intensive Systems, LNCS 5380, pp. 255–265, 2008.

matrix computing, chaos computing, and the entire field of quantum computing; here I will focus on chemical computing.[1]

2 Present State of DNA Computing

While inorganic chemistry was recognized early on as being capable of modelling computation, esp. analogue computation, *DNA computing today seems especially promising, primarily because of the inherent nature of DNA as information carrier*. That is, the specificity of Watson-Crick binding permits encoding a wide variety of signals that can be processed using reactions that are for the most part modularly designed. We examine in which sense DNA-based experiments perform computation and interpret them in terms of conventional mathematical notions of computation. We also examine their commonalities, in particular the question of DNA word design.

DNA computation in its original formulation [5, 6, 7, 8, 9, 10, 11] seeks to employ the massive parallelism inherent in the small scale of molecules to speed up decision problems. The essential property of nucleic acids, specific hybridization (formation of the double helix) [12, 13, 14, 15] is either exploited to encode solutions as long strings of nucleotides, generate large numbers of random strings and check them in a small number of steps, often manual such as PCR (though more reliable detection is now available [16]), or to construct solutions directly through oligonucleotide self-assembly. A number of NP-complete decision problems have been rendered in this fashion [17, 18, 19, 20], and encodings for general computation [21, 22, 23, 24, 25] and combinatorial games [26] have also been proposed. A limitation of the approach is the need for large amounts of nucleic acid [27]; with amounts that could be afforded in the laboratory, or at all, and the low speed of laboratory steps, it has been difficult to outperform electronic computers. Another limitation has been in imperfect specificity of nucleic acid hybridization. Research in this area has ranged from the physico-chemical constraints on usable nucleotide strings (e.g., melting points; secondary structure) to tools for systematic string generation .

Further variations on the theme of DNA computation have included using proteins instead of nucleic acids, for a larger alphabet [28], sophisticated forms of self-assembly [29, 30], to avoid manual operations, and cellular computation in which cells (real or simulated) are viewed as elementary computational elements, with some form of communication among multiple cells [31, 32, 33, 34, 35, 36, 37, 13, 38, 39, 40, 41, 42, 43, 44, 45, 29, 46]. DNA-based self-assembly can also be a vehicle for autonomous fabrication of massively parallel self-organizing processors, which could even be conventional instruction processors [47].

While early on it was believed that DNA computing might be a competitor to electronics in solving hard computational problems, the focus has now shifted to the use of DNA to *compute in environments where it is uniquely capable of operating*, such as in smart drug delivery to individual cells [48, 49].

[1] I will consider experimental approaches; I will not consider chemically (or biochemically) inspired formal computing systems such as membrane computing (P-systems) (Păun) and the gamma formalism (Banâtre).

3 Enzymatic DNA Computing

I have been most directly involved with chemical computing based on DNA enzymes (deoxyribozymes). In this approach to computing—either digital or analog, depending on the interpretation—signals are represented by concentrations of designated molecular species. While such systems can be devised with protein enzymes, we have used smaller DNA enzyme molecules. Deoxyribozymes are enzymes made of DNA that catalyze DNA reactions such as by cleaving a DNA strand into two or ligating two strands into one. Cleaving enzymes (phosphodiesterases) can be modified to include *allosteric regulation sites* to which specific control molecules can bind and so affect the catalytic activity. There is a type of regulation site to which a control molecule must bind before the enzyme can bind to the substrate, thus the control molecule promotes catalytic activity. Another type of regulation site allows the control molecule to alter the conformation of the enzyme's catalytic core, such that even if the substrate has bound to the enzyme, no cleavage occurs; thus this control molecule suppresses or inhibits catalytic activity. An allosterically regulated enzyme can be interpreted as a logic gate, its control molecules as inputs to the gate, and its cleavage products as the outputs. This basic logic gate corresponds to a conjunction, e.g., $a \wedge b \wedge \neg c$, here assuming two allosteric promotory sites and one allosteric inhibitory site, and using a and b as signals encoded by the promotor input molecules and c as a signal encoded by the inhibitor input molecule. In the laboratory, deoxyribozyme logic gates are constructed via a modular design that combines molecular beacon stem-loops [50] with hammerhead-type deoxyribozymes. A gate is active when its catalytic core is intact (not modified by an inhibitory input) and its substrate recognition region is free (owing to the promotive inputs which open the stem-loops), allowing the substrate to bind and be cleaved. Correct functioning of individual gates can be experimentally verified through fluorescent readouts [51].

The gates use oligonucleotides as both inputs and outputs, so cascading gates is possible without external interfaces (such as, e.g., photoelectronics). The inputs are compatible with sensor molecules [52] that could detect cellular disease markers. Final outputs can be tied to release of small molecules. Two gates are coupled *in series* if the product of an "upstream" gate specifically activates a "downstream" gate. All products and inputs (i.e., external signals) must be sufficiently different to minimize the error rates of imperfect oligonucleotide matching, and they must not bind to one another (this is an area of active research). Cascade connections of gates have been experimentally validated (e.g., from an upstream ligase to a downstream phosphodiesterase [53]).

Multiple elementary gates have been constructed, so there is a large number of equivalent ways that any given Boolean function can be realized—equivalent in terms of digital function, but not in speed or cost of realization. For instance, a single four-input gate may be preferable to a cascade with three two-input gates. Clearly, construction of deoxyribozyme logic circuits bears resemblance to traditional low-level logic design, but, perhaps because the technology has not matured, with many more options to explore.

3.1 Simple Enzymatic Circuits

Deoxyribozyme logic gates have been used to build computational devices. A half-adder was achieved by combining three two-input gates in solution [54]. It can be

implemented using an XOR gate for the sum bit and an AND gate for the carry bit. The XOR gate, in turn, is implemented using two ANDNOT gates. The two substrates used are fluorogenically marked with tetramethylrhodamine and with fluorescein, and the activity of the device can be followed by tracking the fluorescence at two distinct wavelengths.

3.2 Enzymatic Game Automata

Using deoxyribozyme logic gates, an automaton for the game of tic-tac-toe has been constructed [55, 56]. To understand how this was achieved, we first briefly examine the structure of that game. A *sequential game* is a game in which players take turns making decisions known as *moves*. A *game of perfect information* is a sequential game in which all the players are informed before every move of the complete state of the game. A *strategy* for a player in a game of perfect information is a plan that dictates what moves that player will make in every possible game state. A *strategy tree* is a (directed, acyclic) graph representation of a strategy. The nodes of the graph represent reachable game states. The edges of the graph represent the opponent's moves. The target node of the edge contains the strategy's response to the move encoded on the edge. A leaf represents a final game state, and can, usually, be labelled either win, lose, or draw. Thus, a path from the root of a strategy tree to one of its leaves represents a game.

In a tree, there is only one path from the root of the tree to each node. This path defines a set of moves made by the players in the game. A player's *move set* at any node is the set of moves made by that player up to that point in a game. For example, a strategy's move set at any node is the set of moves dictated by the strategy along the path from the root to that node. A strategy is said to be *feasible* if, for every pair of nodes in the decision tree for which the opponent's move sets are equal, one of the following two conditions holds: (1) the vertices encode the same decision (i.e., they dictate the same move), or (2) the strategy's move sets are equal. A feasible strategy can be successfully converted into Boolean logic implemented using monotone logic gates, such as the deoxyribozyme logic gates.

In the tic-tac-toe automaton, the following simplifying assumptions are made to reduce the number and complexity of needed molecular species. The automaton moves first and its first move is into the center. Because of symmetry, the first move of the human, which must be either a side move or a corner move, is restricted to one particular corner and one particular side.

The game tree in Figure 1 represents the chosen strategy for the automaton. For example, if the human opponent moves into square 1 following the automaton's opening move into square 5, the automaton responds by moving into square 4 (as indicated on edge 21). If the human then moves into square 6, the automaton responds by moving into square 3 (edge 22). If the human then moves into square 7, the automaton responds by moving into square 2 (edge 23). Finally, if the human then moves into square 8, the automaton responds by moving into square 9, and the game ends in a draw.

This strategy is feasible [57]; therefore, following a conversion procedure, it is possible to reach a set of Boolean formulae that realize it, given in Table 1. The arrangement of deoxyribozyme logic gates corresponding to the above formulae is given in Figure 2.

Table 1. Boolean formulae resulting from the tic-tac-toe game tree

$$o_1 = i_4$$
$$o_2 = (i_6 \wedge i_7 \wedge \neg i_2) \vee (i_7 \wedge i_9 \wedge \neg i_1) \vee (i_8 \wedge i_9 \wedge \neg i_1)$$
$$o_3 = (i_1 \wedge i_6) \vee (i_4 \wedge i_9)$$
$$o_4 = i_1$$
$$o_5 = 1$$
$$o_6 = (i_1 \wedge i_2 \wedge \neg i_6) \vee (i_1 \wedge i_3 \wedge \neg i_6) \vee (i_1 \wedge i_7 \wedge \neg i_6) \vee (i_1 \wedge i_8 \wedge \neg i_6) \vee (i_1 \wedge i_9 \wedge \neg i_6)$$
$$o_7 = (i_2 \wedge i_6 \wedge \neg i_7) \vee (i_6 \wedge i_8 \wedge \neg i_7) \vee (i_6 \wedge i_9 \wedge \neg i_7) \vee (i_9 \wedge i_2 \wedge \neg i_1)$$
$$o_8 = i_9 \wedge i_7 \wedge \neg i_4$$
$$o_9 = (i_7 \wedge i_8 \wedge \neg i_4) \vee (i_4 \wedge i_2 \wedge \neg i_9) \vee (i_4 \wedge i_3 \wedge \neg i_9) \vee (i_4 \wedge i_6 \wedge \neg i_9) \vee (i_4 \wedge i_7 \wedge \neg i_9) \vee (i_4 \wedge i_8 \wedge \neg i_9)$$

This is the initial state of the nine wells of a well-plate in which the automaton is realized in the laboratory.

The play begins when Mg^{2+} ions are added to all nine wells, activating only the deoxyribozyme in well 5, i.e., prompting the automaton to play its first move into the center. After that, the game branches according to the opponent's inputs.

Building this automaton was relatively straightforward in the lab, once the logic was elucidated. The second-generation automaton [56], which plays the full game, and has many more concurrently operating gates that analyze many more inputs, required a great deal of tuning in the laboratory.

4 Computing with Chemical Open Systems and Recurrent Circuits

The first oscillatory chemical reaction was discovered by Belousov in the fifties but for a while remained little known [58]. Once this Belousov-Zhabotinsky reaction became better known and its mechanisms understood [59, 60, 61], it inspired treatments of chemical computation devices, made out of hypothetical large systems of coupled chemical reactions with many stable states [62, 63, 64, 65, 66, 67, 68, 69]; moreover information-theoretic connections were made with Maxwell's daemon [70], and, chaotic behavior having been observed, with unpredictability [71, 72, 73]. Chemical reactions, owing to diffusion, have a spatial component in addition to the temporal. Therefore the oscillatory Belousov-Zhabotinsky reaction gives rise to waves [74]; this was used to implement computation on a prefabricated spatial pattern by wave superposition [75]. Recently an oligonucleotide periodic system was shown [76] (see also [77]).

It has been suggested that computational devices based on chemical kinetics are Turing-equivalent [78], but one must consider the inherently finite number of reactions and molecular species possible [79], and the difficulty of constructing them in practice [80, 81]. Deoxyribozyme logic provides a systematic method for such a construction, and recurrent circuits, including flip-flops and oscillators, have been designed *in silico* on the basis of it [82, 83]. The great cost of DNA material is an obstacle to large-scale experiments at present.

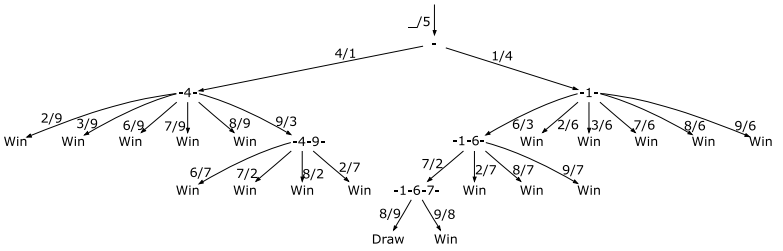


Fig. 1. Game tree for the symmetry-pruned game of tic-tac-toe, drawn as the diagram of a Mealy automaton. Each state is labelled according to the inputs seen on the path to it. Each edge is labelled $a/b(n)$, where b is the output that is activated on input a, and n is the edge identifier.

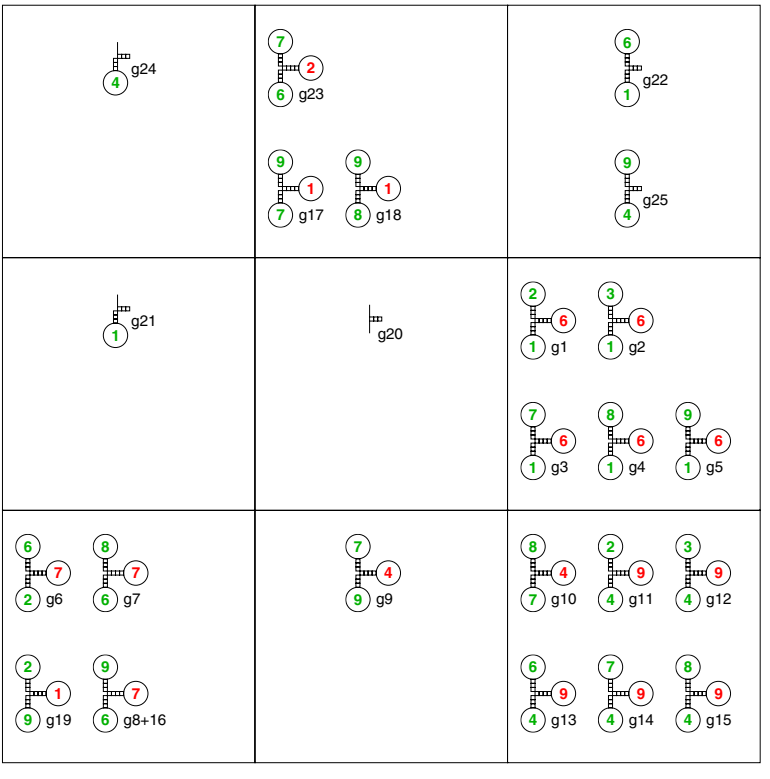


Fig. 2. Realizing a tic-tac-toe automaton using deoxyribozyme logic. The center well contains a consitutively active deoxyribozyme. Each of the eight remaining wells contains a number of deoxyribozyme logic gates as indicated.

Beyond the laboratory, certain naturally occurring coupled biochemical reactions of this kind have been identified (in signalling and metabolic pathways) and their computational nature has been suggested [84, 85, 86, 87, 88, 14]. We note in passing that

there have also been early suggestions for irreversible logic gates as models of chemical kinetics [89] inspired by observations of biological systems [90]. Recently, there has been interest in natural computing, in particular to multicellular slime molds that exhibit multiple life phases and dynamically adaptive optimization behaviors that can be interpreted as computation, such as in the *Dictyostelium* and the *Physarum*. I am exploring computational aspects of much smaller agents, molecular robotic assemblies [91].

5 Assessment

(Bio)chemical computing is not ready for prime time as a means of creating practical computing devices for everyday use. It is unlikely it will ever supplant electronic computers, but it may soon find niche uses: (1) Analogue computation in those cases where the structure of the problem being explored parallels the structure of the (bio)chemical system. (2) Autonomous computation in environments where electronics cannot be deployed.

Lack of programmability, modular design, and error correction (or estimation for the analogue case) remain serious deficiencies—and new research should resolve these issues.

Acknowledgements. Thanks are due to Clint Morgan, Joseph Farfel, Ben Andrews, Jenny Sager, and Cris Moore (University of New Mexico) and Tiffany Mitchell, Joanne Macdonald, and Milan Stojanovic (Columbia University). This material is based upon work supported by the National Science Foundation under grants 0238027, 0324845, and 0533065.

References

1. Ganusov, I., Burtscher, M.: Future execution: A prefetching mechanism that uses multiple cores to speed up single threads. ACM Transactions on Architecture and Code Optimization 3(4), 424–449 (2006)
2. Abelson, H., Allen, D., Coore, D., Hanson, C., Homsy, G., Knight Jr., T.F., Nagpal, R., Rauch, E., Sussman, G.J., Weiss, R.: Amorphous computing. Communications of the ACM 43(5), 74–82 (2000)
3. Goldstein, S.C., Rosewater, D.: Digital logic using molecular electronics. In: IEEE International Solid-State Circuits Conference, San Francisco, CA, p. 12.5 (February 2002)
4. Gruau, F., Lhuillier, Y., Reitz, P., Temam, O.: Blob computing. In: Computing Frontiers 2004 ACM SIGMicro (June 2004)
5. Adleman, L.M.: Molecular computation of solutions to combinatorial problems. Science 266(5187), 1021–1024 (1994)
6. Deaton, R.J., Garzon, M., Rose, J.A., Franceschetti, D.R., Stevens Jr., S.E.: DNA computing: A review. Fundamenta Informaticae 35(1-4), 231–245 (1998)
7. Lipton, R.J.: DNA solution of hard computational problems. Science 268, 542–545 (1995)
8. Ruben, A.J., Landweber, L.F.: Timeline: The past, present and future of molecular computing. Nature Reviews Molecular Cell Biology 1, 69–72 (2000)
9. Wang, L., Liu, Q., Corn, R.M., Condon, A.E., Smith, L.M.: Multiple word DNA computing on surfaces. Journal of the American Chemical Society 122(31), 7435–7440 (2000)

10. Winfree, E.: On the computational power of DNA annealing and ligation. In: Lipton, Baum (eds.) [93], pp. 199–221
11. Winfree, E.: Complexity of restricted and unrestricted models of molecular computation. In: Lipton, Baum (eds.) [93], pp. 187–198
12. Watson, J.D., Crick, F.H.C.: A structure for deoxyribose nucleic acid. Nature 171, 737 (1953)
13. LaBean, T.H., Yan, H., Kopatsch, J., Liu, F., Winfree, E., Reif, J.H., Seeman, N.C.: Construction, analysis, ligation, and self-assembly of DNA triple crossover complexes. Journal of the American Chemical Society 122, 1848–1860 (2000)
14. Watson, J.D., Hopkins, N.H., Roberts, J.W., Steitz, J.A., Weiner, A.M.: Molecular Biology of the Gene, 4th edn., Benjamin/Cummings, Menlo Park, CA (1988)
15. Winfree, E., Liu, F., Wenzler, L.A., Seeman, N.C.: Design and self-assembly of two-dimensional DNA crystals. Nature 394, 539–544 (1998)
16. Wang, L., Hall, J.G., Lu, M., Liu, Q., Smith, L.M.: A DNA computing readout operation based on structure-specific cleavage. Nature Biotechnology 19, 1053–1059 (2001)
17. Braich, R.S., Chelyapov, N., Johnson, C., Rothemund, P.W.K., Adleman, L.: Solution of a 20-variable 3-SAT problem on a DNA computer. Science 296, 499–502 (2002)
18. Morimoto, N., Arita, M., Suyama, A.: Solid phase DNA solution to the Hamiltonian path problem. In: Rubin, Wood (eds.) [92], pp. 193–206
19. Ouyang, Q., Kaplan, P.D., Liu, S., Libchaber, A.: DNA solution of the maximal clique problem. Science 278, 446–449 (1997)
20. Pirrung, M.C., Connors, R.V., Odenbaugh, A.L., Montague-Smith, M.P., Walcott, N.G., Tollett, J.J.: The arrayed primer extension method for DNA microchip analysis. Molecular computation of satisfaction problems. Journal of the American Chemical Society 122, 1873–1882 (2000)
21. Garzon, M., Gao, Y., Rose, J.A., Murphy, R.C., Deaton, R.J., Franceschetti, D.R., Stevens Jr., S.E.: In vitro implementation of finite-state machines. In: Wood, D., Yu, S. (eds.) WIA 1997. LNCS, vol. 1436, pp. 56–74. Springer, Heidelberg (1998)
22. Guarnieri, F., Fliss, M., Bancroft, C.: Making DNA add. Science 273, 220–223 (1996)
23. Hug, H., Schuler, R.: DNA-based parallel computation of simple arithmetic. In: Jonoska, N., Seeman, N.C. (eds.) DNA 2001. LNCS, vol. 2340. Springer, Heidelberg (2002)
24. Mao, C., LaBean, T.H., Reif, J.H., Seeman, N.C.: Logical computation using algorithmic self-assembly of DNA triple-crossover molecules. Nature 407, 493–496 (2000); Erratum. Nature 408, 750 (2000)
25. Rothemund, P.W.K., Winfree, E.: The program-size complexity of self-assembled squares. In: STOC 2000: The Thirty-Second Annual ACM Symposium on Theory of Computing (May 2000)
26. Faulhammer, D., Cukras, A.R., Lipton, R.J., Landweber, L.F.: Molecular computation: RNA solutions to chess problems. Proceedings of the National Academy of Sciences of the USA (PNAS) 97(4), 1385–1389 (2000), http://www.pnas.org/cgi/content/full/97/4/1385/DC1
27. Hartmanis, J.: On the weight of computation. Bulletin of the EATCS 55, 136–138 (1995)
28. Hug, H., Schuler, R.: Strategies for the development of a peptide computer. Bioinformatics 17(4), 364–368 (2001)
29. Winfree, E.: Simulations of computing by self-assembly. In: Kari, L., Rubin, H., Wood, D.H. (eds.) DNA Based Computers IV, DIMACS Workshop 1998 (University of Pennsylvania: Philadelphia, PA, Biosystems, vol. 52(1-3), pp. 213–242. Elsevier, Amsterdam (1998)
30. Rothemund, P.W.K.: Folding DNA to create nanoscale shapes and patterns. Nature 440, 297–302 (2006)
31. Basu, S., Karig, D., Weiss, R.: Engineering signal processing in cells: Towards molecular concentration band detection. In: Hagiya, M., Ohuchi, A. (eds.) DNA 2002. LNCS, vol. 2568. Springer, Heidelberg (2003)

32. Conrad, M.: On design principles for a molecular computer. Communications of the ACM 28(3), 464–480 (1985)
33. Guet, C.C., Elowitz, M.B., Wang, W., Leibler, S.: Combinatorial synthesis of genetic networks. Science 296, 1466–1470 (2002)
34. Hayes, B.: Computing comes to life. American Scientist 89, 204–208 (2001)
35. Ji, S.: The cell as the smallest DNA-based molecular computer. BioSystems 52, 123–133 (1999)
36. Knight Jr., T.F., Sussman, G.J.: Cellular gate technology. In: Proceedings UMC 1998, First International Conference on Unconventional Models of Computation (1998)
37. LaBean, T.H., Winfree, E., Reif, J.H.: Experimental progress in computation by self-assembly of DNA tilings. In: Winfree, E., Gifford, D.K. (eds.) DNA Based Computers V, DIMACS Workshop 1999 (MIT: Cambridge, MA). Series in Discrete Mathematics and Theoretical Computer Science, vol. 54, pp. 123–140. American Mathematical Society (2000)
38. Landweber, L.F., Kari, L.: The evolution of cellular computing: nature's solution to a computational problem. BioSystems 52(1–3), 3–13 (1999)
39. Landweber, L.F., Kuo, T.-C., Curtis, E.A.: Evolution and assembly of an extremely scrambled gene. Proceedings of the National Academy of Sciences of the USA 97(7), 3298–3303 (2000)
40. Reif, J.H.: Parallel biomolecular computation. In: Rubin, Wood (eds.) [92], pp. 217–254
41. Saylor, G.: Construction of genetic logic gates for biocomputing. In: 101st General Meeting of the American Society for Microbiology (2001)
42. Weiss, R.: Cellular Computation and Communication using Engineered Genetic Regulatory Networks. PhD thesis, Massachusetts Institute of Technology (September 2001)
43. Weiss, R., Basu, S.: The device physics of cellular logic gates. In: First Workshop on Non-Silicon Computing (February 2002)
44. Weiss, R., Homsy, G., Nagpal, R.: Programming biological cells. Technical report, MIT Laboratory for Computer Science and Artificial Intelligence (1998)
45. Weiss, R., Homsy, G.E., Knight Jr., T.F.: Towards in vivo digital circuits. In: DIMACS Workshop on Evolution as Computation (January 1999)
46. Winfree, E., Yang, X., Seeman, N.C.: Universal computation via self-assembly of DNA: Some theory and experiments. In: Landweber, L.F., Baum, E.B. (eds.) DNA Based Computers II, DIMACS Workshop 1996 (Princeton University: Princeton, NJ). Series in Discrete Mathematics and Theoretical Computer Science, vol. 44, pp. 191–213. American Mathematical Society (1999), http://www.dna.caltech.edu/Papers/self-assem.errata
47. Patwardhan, J., Johri, V., Dwyer, C., Lebeck, A.R.: A defect tolerant self-organizing nanoscale simd architecture. In: Proceedings of the Twelth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XII (2006)
48. Cox, J.C., Ellington, A.D.: DNA computation function. Current Biology 11(9), R336 (2001)
49. Yurke, B., Mills Jr., A.P., Cheng, S.L.: DNA implementation of addition in which the input strands are separate from the operator strands. BioSystems 52(1–3), 165–174 (1999)
50. Stojanovic, M.N., de Prada, P., Landry, D.W.: Catalytic molecular beacons. Chem. Bio. Chem. 2(6), 411–415 (2001)
51. Stojanovic, M.N., Mitchell, T.E., Stefanovic, D.: Deoxyribozyme-based logic gates. Journal of the American Chemical Society 124(14), 3555–3561 (2002)
52. Stojanovic, M.N., Kolpashchikov, D.: Modular aptameric sensors. Journal of the American Chemical Society 126(30), 9266–9270 (2004)
53. Stojanovic, M.N., Semova, S., Kolpashchikov, D., Morgan, C., Stefanovic, D.: Deoxyribozyme-based ligase logic gates and their initial circuits. Journal of the American Chemical Society 127(19), 6914–6915 (2005)

54. Stojanovic, M.N., Stefanovic, D.: Deoxyribozyme-based half adder. Journal of the American Chemical Society 125(22), 6673–6676 (2003)
55. Stojanovic, M.N., Stefanovic, D.: A deoxyribozyme-based molecular automaton. Nature Biotechnology 21(9), 1069–1074 (2003)
56. Macdonald, J., Li, Y., Sutovic, M., Lederman, H., Pendri, K., Lu, W., Andrews, B.L., Stefanovic, D., Stojanovic, M.N.: Medium scale integration of molecular logic gates in an automaton. Nano Letters 6(11), 2598–2603 (2006)
57. Andrews, B.: Games, strategies, and boolean formula manipulation. Master's thesis, University of New Mexico (December 2005)
58. Epstein, I.R., Pojman, J.A.: An Introduction to Nonlinear Chemical Dynamics. Oxford University Press, New York (1998)
59. Field, R.J., Körös, E., Noyes, R.: Oscillations in chemical systems. II. Thorough analysis of temporal oscillation in the bromate-cerium-malonic acid system. Journal of the American Chemical Society 94, 8649–8664 (1972)
60. Noyes, R., Field, R.J., Körös, E.: Oscillations in chemical systems. I. Detailed mechanism in a system showing temporal oscillations. Journal of the American Chemical Society 94, 1394–1395 (1972)
61. Tyson, J.J.: The Belousov-Zhabotinskii Reaction. Lecture Notes in Biomathematics, vol. 10. Springer, Berlin (1976)
62. Hjelmfelt, A., Ross, J.: Chemical implementation and thermodynamics of collective neural networks. Proceedings of the National Academy of Sciences of the USA 89(1), 388–391 (1992)
63. Hjelmfelt, A., Ross, J.: Pattern recognition, chaos, and multiplicity in neural networks of excitable systems. Proceedings of the National Academy of Sciences of the USA 91(1), 63–67 (1994)
64. Hjelmfelt, A., Schneider, F.W., Ross, J.: Pattern recognition in coupled chemical kinetic systems. Science 260, 335–337 (1993)
65. Hjelmfelt, A., Weinberger, E.D., Ross, J.: Chemical implementation of neural networks and Turing machines. Proceedings of the National Academy of Sciences of the USA 88(24), 10983–10987 (1991)
66. Hjelmfelt, A., Weinberger, E.D., Ross, J.: Chemical implementation of finite-state machines. Proceedings of the National Academy of Sciences of the USA 89(1), 383–387 (1992)
67. Laplante, J.-P., Pemberton, M., Hjelmfelt, A., Ross, J.: Experiments on pattern recognition by chemical kinetics. The Journal of Physical Chemistry 99(25), 10063–10065 (1995)
68. Rössler, O.E., Seelig, F.F.: A Rashevsky-Turing system as a two-cellular flip-flop. Zeitschrift für Naturforschung 27b, 1444–1448 (1972)
69. Seelig, F.F., Rössler, O.E.: Model of a chemical reaction flip-flop with one unique switching input. Zeitschrift für Naturforschung 27b, 1441–1444 (1972)
70. Szilard, L.: Über die Entropieverminderung in einem thermodynamischen System bei Eingriffen intelligenter Wesen. Zeitschrift für Physik 53, 840–856 (1929)
71. Matías, M.A., Güémez, J.: On the effects of molecular fluctuations on models of chemical chaos. Journal of Chemical Physics 102(4), 1597–1606 (1995)
72. Moore, C.: Unpredictability and undecidability in dynamical systems. Physical Review Letters 64(20), 2354–2357 (1990)
73. Wolfram, S.: Undecidability and intractability in theoretical physics. Physical Review Letters 54(8), 735–738 (1985)
74. Winfree, A.T.: Spiral waves of chemical activity. Science 175, 634–635 (1972)
75. Steinbock, O., Kettunen, P., Showalter, K.: Chemical wave logic gates. Journal of Physical Chemistry 100, 18970–18975 (1996)

76. Yurke, B., Turberfield, A.J., Mills Jr., A.P., Neumann, J.L.: A molecular machine made of and powered by DNA. In: The 2000 March Meeting of the American Physical Society (March 2000)
77. Magnasco, M.O.: Molecular combustion motors. Physical Review Letters 72(16), 2656–2659 (1994)
78. Magnasco, M.O.: Chemical kinetics is Turing universal. Physical Review Letters 78(6), 1190–1193 (1997)
79. Homsy, G.E.: Performance limits on biochemical computation. Technical report, MIT Artificial Intelligence Laboratory (2000)
80. Hiratsuka, M., Aoki, T., Higuchi, T.: Enzyme transistor circuits for reaction-diffusion computing. IEEE Transactions on Circuits and systems—I: Fundamental Theory and Applications 46(2), 294–303 (1999)
81. Hiratsuka, M., Aoki, T., Morimitsu, H., Higuchi, T.: Implementation of reaction-diffusion cellular automata. IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications 49(1), 10–16 (2002)
82. Morgan, C., Stefanovic, D., Moore, C., Stojanovic, M.N.: Building the components for a biomolecular computer. In: Ferretti, C., Mauri, G., Zandron, C. (eds.) Preliminary Proceedings of the 10th International Workshop on DNA-Based Computers, DNA 2004. University of Milano-Bicocca, Milan (2004)
83. Farfel, J., Stefanovic, D.: Towards practical biomolecular computers using microfluidic deoxyribozyme logic gate networks. In: Carbone, A., Daley, M., Kari, L., McQuillan, I., Pierce, N. (eds.) Preliminary Proceedings of the 11th International Workshop on DNA-Based Computers, DNA 2005, pp. 221–232. University of Western Ontario, London (2005)
84. Alberts, B., Bray, D., Johnson, A., Lewis, J., Raff, M., Roberts, K., Walter, P.: Essential Cell Biology: An Introduction to the Molecular Biology of the Cell. Garland, New York (1998)
85. Arkin, A., Ross, J.: Computational functions in biochemical reaction networks. Biophysical Journal 67(2), 560–578 (1994)
86. Goldbeter, A.: Biochemical Oscillations and Cellular Rhythms: The molecular bases of periodic and chaotic behaviour. Cambridge University Press, Cambridge (1996)
87. Okamoto, M., Hayashi, K.: Dynamic behavior of cyclic enzyme systems. Journal of Theoretical Biology 104, 591–598 (1983)
88. Okamoto, M., Sakai, T., Hayashi, K.: Switching mechanism of a cyclic enzyme system: Role as a "chemical diode". BioSystems 21(1), 1–11 (1987)
89. Sugita, M.: Functional analysis of chemical systems *in vivo* using a logical circuit equivalent. II. The idea of a molecular automaton. Journal of Theoretical Biology 4, 179–192 (1963)
90. Fukuda, N., Sugita, M.: Mathematical analysis of metabolism using an analogue computer: I. Isotope kinetics of iodine metabolism in the thyroid gland. Journal of Theoretical Biology 1, 440–459 (1961)
91. Pei, R., Taylor, S.K., Stefanovic, D., Rudchenko, S., Mitchell, T.E., Stojanovic, M.N.: Behavior of polycatalytic assemblies in a substrate-displaying matrix. Journal of the American Chemical Society 128(39), 12693–12699 (2006)
92. Rubin, H., Wood, D.H. (eds.): DNA Based Computers III, DIMACS Workshop 1997 (University of Pennsylvania: Philadelphia, PA). Series in Discrete Mathematics and Theoretical Computer Science, vol. 48. American Mathematical Society (1999)
93. Lipton, R.J., Baum, E.B.: DNA Based Computers, DIMACS Workshop 1995 (Princeton University: Princeton, NJ). Series in Discrete Mathematics and Theoretical Computer Science, vol. 27. American Mathematical Society (1996)

Author Index