# Executable Modeling of Deployment Decisions for Resource-Aware Distributed Applications

Doctoral Dissertation by

Silvia Lizeth Tapia Tarifa

Submitted to the
Faculty of Mathematics and Natural Sciences at the University of Oslo
for the degree Philosophiae Doctor in Computer Science

*Date of submission:* February 2014
*Date of public defense:* May 2014

Precise Modeling and Analysis Group
Department of Informatics
University of Oslo
Norway

Oslo, February 2014

**© Silvia Lizeth Tapia Tarifa, 2014**

*Series of dissertations submitted to the*
*Faculty of Mathematics and Natural Sciences, University of Oslo*
*No. 1484*

ISSN 1501-7710

All rights reserved. No part of this publication may be
reproduced or transmitted, in any form or by any means, without permission.

Cover: Inger Sandved Anfinsen.
Printed in Norway: AIT Oslo AS.

Produced in co-operation with Akademika Publishing.
The thesis is produced by Akademika Publishing merely in connection with the
thesis defence. Kindly direct all inquiries regarding the thesis to the copyright
holder or the unit which grants the doctorate.

# Abstract

The increasing popularity of virtualized services and cloud computing, offering elastic and scalable computing resources challenges software engineering methods by asking a number of new questions: How can we integrate deployment-specific information in the overall design of software applications? How can we express and compare deployment decisions in the design phase, so that performance diagnosis can happen early in the software development cycle? How do we design scalable applications?

This thesis proposes a *methodology* for the modeling and analysis of object-oriented distributed applications that are able to adapt to changes in client traffic, so that they can keep desired levels of performance. As a starting point for developing our methodology, we take Core ABS, an abstract, behavioral specification language that targets the executable modeling of concurrent, distributed and object-oriented systems. We extend Core ABS with deployment-specific information such as deployment architectures, flexible application-specific schedulers, deadlines at the application level, resource management primitives, and user-defined resource costs. The extensions are developed following conventions of formal methods, combining rigorous definitions and formal semantics with a user-friendly Java-like syntax and tool support. By exploiting these language extensions, our methodology makes it possible to compare at the modeling level how a software behaves under different deployment choices. Such comparisons allow a better understanding of the trade-offs from different deployment choices, consequently better design decisions for distributed applications can be made early in the software development life cycle.

By integrating resource management and deployment decisions in the design phase, our methodology has the potential to improve the software engineering process for virtualized and scalable software applications running in the cloud, both with respect to quality and with respect to development and deployment cost.

## *Acknowledgments*

I am deeply grateful for all those who made it possible for me to reach this point on my academic path. Now I am at the end of my Ph.D. studies and I am certainly not the same person as when I started. All these years in academia have helped me to grow not only in knowledge but also in other aspects of my personal life.

I would like to express my deep gratitude to my main supervisor Einar Broch Johnsen. I am grateful for his priceless time and help whenever I needed, constant encouragement, patient guidance, fruitful discussions, constructive and constant co-operations, optimism in reaching various deadlines, and for all the knowledge I have learned from him. I am also grateful for his great friendship and for teaching me various things of life, like to appreciate a good bottle of wine and a good cup of coffee. I would also like to express my special thanks to Rudolf Schlatte for his constant cooperation and active collaboration in various papers; we have shared a lot of working and non-working time, and I am very grateful for that. I would also like to thank Olaf Owe for his advice, support, wisdom, collaboration and knowledge, and Martin Steffen for his interesting discussions, passionate teaching and help whenever I needed.

I would like to offer my deep thanks to all my colleagues, ex-colleagues and good friends in the PMA group. It has been a wonderful time together, both inside and outside the office. I enjoyed the working environment and I always felt very welcome in the PMA family. In particular, thanks to Crystal Chang Din for being a wonderful office mate.

I am grateful for the assistance provided by the technical support and the administration staff at the Department of Informatics, University of Oslo. I would also like to thank everybody involved in the HATS project, in particular Reiner Hähnle, Frank de Boer and Elvira Albert. My thanks also go to all researchers and professors who have crossed my path as friends and colleagues over the years, in particular Chris George, Abigail Parisaca and Wilber Ramos.

I would also like to thank my parents, brothers and extended family. Words cannot express how grateful I am to my parents Patricia Tarifa de Tapia and Walter Tapia Salas, and to my brothers Fabricio Tapia Tarifa and Alex Tapia Tarifa. They have always encouraged and supported my goals, and they have always been with me despite the distance. Finally I would like to thank my friends for their company during these years, in particular all my Peruvian friends in Europe, especially Liliana Mamani and Jessica Vargas.

The work in this thesis has been done at the Department of Informatics of the University of Oslo, supported by a grant from the Faculty of Mathematics and Natural Science of the University of Oslo. Additional funding has been provided by the HATS project and by the Department of Informatics.

# *Contents*

**Abstract** iii

**Acknowledgments** v

**I Overview** 1

**1 Introduction** 3

1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.2 Research Questions . . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.3 Structure of this Thesis . . . . . . . . . . . . . . . . . . . . . . . . 6

**2 The ABS Language and its Underlying Concepts** 9

2.1 Concurrent Languages . . . . . . . . . . . . . . . . . . . . . . . . 9

2.2 Object-Oriented Languages and Concurrency . . . . . . . . . . . . . . . 11

2.3 Formal Modeling Languages and Concurrency . . . . . . . . . . . . . . 14

2.4 The ABS Language . . . . . . . . . . . . . . . . . . . . . . . . . 15

2.4.1 The Functional and Imperative Layers of Core ABS . . . . . . . 16

2.4.2 Design Choices of Core ABS . . . . . . . . . . . . . . . . . . 17

2.5 The HATS Project . . . . . . . . . . . . . . . . . . . . . . . . . . 18

**3 Overview of the Research Papers** 19

3.1 Paper 1: User-defined Schedulers and Real-Time . . . . . . . . . . . . . 19

3.2 Paper 2: Deployment Architectures and Resource Consumption . . . . 20

3.3 Paper 3: Resource-Awareness and Cloud Infrastructures . . . . . . . . . 22
3.4 Paper 4: Deployment Architectures and Worst-Case Cost Bounds . . . 23
3.5 Additional Publications . . . . . . . . . . . . . . . . . . . . . . . . . . 23

**4 Conclusions 27**
4.1 Summary of Contributions . . . . . . . . . . . . . . . . . . . . . . . . 27
4.2 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

**II Research Papers 41**

**5 Paper 1: User-defined Schedulers and Real-Time 43**
5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
5.2 Real-Time ABS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
5.2.1 The Functional Level of Real-Time ABS . . . . . . . . . . . . . 48
5.2.2 The Concurrent Object Level of Real-Time ABS . . . . . . . . . 49
5.3 Scheduling Strategies in Real-Time ABS . . . . . . . . . . . . . . . . . 52
5.3.1 General Scheduling Policies . . . . . . . . . . . . . . . . . . . . 54
5.3.2 Conditional Scheduler . . . . . . . . . . . . . . . . . . . . . . . 56
5.3.3 Scheduling Annotations in Real-Time ABS . . . . . . . . . . . . 57
5.3.4 Monitors with Signal and Continue Discipline . . . . . . . . . . 57
5.4 Semantics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
5.4.1 Runtime Configurations . . . . . . . . . . . . . . . . . . . . . . 59
5.4.2 A Reduction System for Expressions . . . . . . . . . . . . . . . 61
5.4.3 A Transition System for Timed Configurations . . . . . . . . . . 62
5.5 Case Studies and Simulation Results . . . . . . . . . . . . . . . . . . . 67
5.6 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69

**6 Paper 2: Deployment Architectures and Resource Consumption 75**
6.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
6.2 Modeling Timed Behavior in Real-Time ABS . . . . . . . . . . . . . . . 78
6.2.1 The Functional Layer of Real-Time ABS . . . . . . . . . . . . . 79

6.2.2 The Imperative Layer of Real-Time ABS . . . . . . . . . . . . . 82
6.2.3 Explicit and Implicit Time in Real-Time ABS . . . . . . . . . . . 84
6.3 Modeling Deployment Architectures . . . . . . . . . . . . . . . . . . . . . 85
6.3.1 Deployment Components . . . . . . . . . . . . . . . . . . . . . . . . . 86
6.3.2 Resource Consumption . . . . . . . . . . . . . . . . . . . . . . . . . . 88
6.3.3 The Deployment Layer of Real-Time ABS . . . . . . . . . . . . . 89
6.4 Example: A Client-Server System . . . . . . . . . . . . . . . . . . . . . . 90
6.5 Example: Implementing Object Migration to Mitigate Overload . . . . 93
6.6 Example: Load Balancing via Resource Transfer . . . . . . . . . . . . . 96
6.7 Semantics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
6.7.1 Runtime Configurations . . . . . . . . . . . . . . . . . . . . . . . . . 99
6.7.2 The Timed Evaluation of Expressions . . . . . . . . . . . . . . . . 102
6.7.3 A Transition System for Timed Configurations . . . . . . . . . . 103
6.8 Related and Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
6.9 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113

**7 Paper 3: Resource-Awareness and Cloud Infrastructures 123**
7.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124
7.2 Abstract Behavioral Specification with Real-Time ABS . . . . . . . . . 125
7.2.1 Modeling Timed Behavior in ABS . . . . . . . . . . . . . . . . . . . 125
7.2.2 Modeling Deployment Architectures in Real-Time ABS . . . . . . . . . . . . . . . . . . . . . . . . . . 127
7.3 Resource Management and Cloud Provisioning . . . . . . . . . . . . . . 128
7.4 Case Study: The Montage Toolkit . . . . . . . . . . . . . . . . . . . . . . 131
7.4.1 The Problem Description . . . . . . . . . . . . . . . . . . . . . . . . 131
7.4.2 A Model of the Montage Workflow in Real-Time ABS . . . . . . 133
7.4.3 Simulation Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . 134
7.5 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 136
7.6 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 138

**8 Paper 4: Deployment Architectures and Worst-Case Cost Bounds 145**

8.1 Introduction . . . 146

8.2 A Language for Distributed Concurrent Objects . . . 147

8.2.1 Operational Semantics . . . 151

8.3 Worst-Case Cost Bounds . . . 153

8.4 Deployment Components . . . 156

8.5 Simulation and Experimental Results . . . 158

8.6 Related Work . . . 160

8.7 Discussion . . . 160

# *List of Figures*

***5 Paper 1: User-defined Schedulers and Real-Time*** ***43***

*5.1 Syntax for the functional level of Real-Time ABS* . . . . . . . . . . . . 47
*5.2 Syntax for the concurrent object level of Real-Time ABS* . . . . . . . . 50
*5.3 Runtime syntax* . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
*5.4 The evaluation of functional expressions* . . . . . . . . . . . . . . . . . 61
*5.5 The semantics of Real-Time ABS* . . . . . . . . . . . . . . . . . . . . 63
*5.6 The semantics of Real-Time ABS* . . . . . . . . . . . . . . . . . . . . 64
*5.7 Functions controlling the advancement of time* . . . . . . . . . . . . . . 65
*5.8 A model of photo and video processing* . . . . . . . . . . . . . . . . . . 66
*5.9 Simulation results for Example 8* . . . . . . . . . . . . . . . . . . . . 68
*5.10 Extending Fig. 5.8 with an application-specific scheduler* . . . . . . . . 69
*5.11 Application-specific scheduling for the server example* . . . . . . . . . . 70

***6 Paper 2: Deployment Architectures and Resource Consumption*** ***75***

*6.1 The Layers of the Real-Time ABS* . . . . . . . . . . . . . . . . . . . . 78
*6.2 Syntax for the functional layer of Real-Time ABS* . . . . . . . . . . . . 79
*6.3 Syntax for the imperative layer of Real-Time ABS* . . . . . . . . . . . . 84
*6.4 Specification of resources and the interface of deployment components* . 86
*6.5 A deployment architecture in Real-Time ABS* . . . . . . . . . . . . . . 88
*6.6 Syntax extension for the deployment layer* . . . . . . . . . . . . . . . . 89
*6.7 A session-oriented server model in Real-Time ABS* . . . . . . . . . . . . 91

6.8 *Deployment environment and client model of the web shop example* . . 92

6.9 *Number of total requests and successful orders, depending on the number of clients and resources* . . . . . . . . . . . . . . . . . . . . . . . . . 93

6.10 *An agent which performs load balancing* . . . . . . . . . . . . . . . . . 94

6.11 *Self-monitoring session objects* . . . . . . . . . . . . . . . . . . . . . . . 95

6.12 *Simulation results for the load balancing strategies* . . . . . . . . . . . 96

6.13 *The telephony and SMS services* . . . . . . . . . . . . . . . . . . . . . . 97

6.14 *The Handset class* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98

6.15 *A resource reallocation strategy and deployment configuration* . . . . . 100

6.16 *Simulation of "New Year's Eve" behavior* . . . . . . . . . . . . . . . . . 101

6.17 *Runtime syntax* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102

6.18 *The evaluation of functional expressions* . . . . . . . . . . . . . . . . . 103

6.19 *Semantics with deployment components and resource consumption (1)* . 105

6.20 *Semantics with deployment components and resource consumption (2)* . 106

6.21 *Functions controlling the advancement of time* . . . . . . . . . . . . . . 110

***7 Paper 3: Resource-Awareness and Cloud Infrastructures 123***

7.1 *Interaction between a client application and the cloud provider* . . . . . 129

7.2 *The CloudProvider class in Real-Time ABS* . . . . . . . . . . . . . . . 130

7.3 *The modules of the Montage case study* . . . . . . . . . . . . . . . . . 132

7.4 *Montage abstract workflow* . . . . . . . . . . . . . . . . . . . . . . . . . 133

7.5 *CalcServer interface and class in Real-Time ABS* . . . . . . . . . . . . . 134

7.6 *The ApplicationServer interface and class (abridged)* . . . . . . . . . . . 135

7.7 *Montage: Execution costs and times of simulation* . . . . . . . . . . . . 136

***8 Paper 4: Deployment Architectures and Worst-Case Cost Bounds 145***

8.1 *ABS syntax for the functional level* . . . . . . . . . . . . . . . . . . . . 148

8.2 *ABS syntax for the concurrent object level.* . . . . . . . . . . . . . . . . 149

8.3 *ABS Semantics* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 153

8.4 *Operational semantics for resource-constrained deployment components* 158

*8.5 Resource-aware assignment rule* . . . . . . . . . . . . . . . . . . . . . . . . 159
*8.6 Final and peak memory use* . . . . . . . . . . . . . . . . . . . . . . . . . . 159

# Part I

# Overview

CHAPTER 1

# Introduction

## 1.1 Motivation

*Software engineering* is the field of computer science that deals with the development of software. The development of software follows a development life cycle model composed of several phases, including requirement, design, implementation, integration and deployment. These phases can be interleaved to fit into traditional (e.g., waterfall), or iterative and incremental (e.g., Agile [1], RAD [14]) development methods [33, 65].

*Modeling* and *analysis* are techniques used to specify and validate software applications so that stakeholders can better understand the system being developed even from early phases in the development process. During the software development process, models provide abstractions of the target applications and provide connections between various phases, e.g., requirement and design.

Models typically cover [65] (1) an *external perspective* on the system to be developed, which describes its interaction with the system's environment; (2) a *behavioral perspective*, which describes the behavior of the system; and (3) a *structural perspective*, which describes the logical architecture of the system and the structure of the data provided in the system. Object-oriented modeling techniques, such as UML [16], usually capture both the behavioral and the structural architecture perspective in the same model.

Ideally, an *application's system architecture model* includes a set of principal *design decisions* made about the system in order to deliver a desired level of service quality [66]. Part of these design decisions comprise general policies on how the application will fulfill *functional* and *non-functional* requirements. Examples of non-functional requirements for applications include an acceptable level of performance in their services, dynamic adaptation of their service capacity to a variable number

of users on-demand, adaptability to different computing infrastructures, and effective usage of their resources (e.g., cpu, memory, bandwidth).

Application developers have made significant progress in modeling and analyzing functional requirements and building architectures that meet them, but it is still a challenge how to deal systematically with non-functional requirements when architecting systems [19]. For example, different characteristics of the hardware infrastructure which are decided during the deployment phase (e.g., the speed of the server and the bandwidth of the network) influence how well an application is able to meet its non-functional requirements related to performance. Looking at the current state of software development literature, there is not a clear link between the deployment phase and the early design of performance requirements [33, 65].

Unlike standalone applications which operate on single devices and in predictable environments, distributed applications operate on heterogenous devices, possibly geographically spread, and interconnected by a communication network. Distributed applications may serve an unknown number of end-users, consequently they can suffer from unexpected changes in their workload. Deploying critical components of a distributed application with possible heavy workloads on devices with limited resources, could negatively impact the performance of the overall application as much as an incorrect implementation of any of its critical functional requirements.

Traditional and iterative software development methods focus on performance problems late in the software development process [12, 70]. These problems may be so severe that they can require considerable changes in design at the software architecture level and, in the worst cases, they can even impact the requirement level [12]. The most common approach to meet performance requirements is empirical, based on previous experiences, and applies methods such as testing, tuning, and diagnosis late in the software development cycle, when the application under development can be executed and measured [70]. Yet, it is not uncommon that companies discover that the performance of their applications is below expectations once the applications are in operation. To increase the chances of successfully meeting performance requirements, it is a hypothesis of this thesis that *it is advantageous to include deployment decisions in the design phase*, to make performance diagnosis available early in the software development cycle.

With the evolution of hardware virtualization, modern systems are increasingly being deployed on large-scale distributed systems offering virtualized scalable services, such as cloud infrastructures. A *cloud* is a parallel and distributed system consisting of a collection of interconnected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources [20]. Clouds are clearly the next-generation of computing resources, with data centers offering services to software applications. Service provisioning in a cloud is based on negotiations between the service provider and consumers. These services are offered in a way which makes it possible to virtualize the deployment of resources based on a pay-per-use

model that provides an *elastic* and *scalable* amount of resources on-demand in the cloud [20].

To take advantage of the cloud's elasticity and scalability, applications should be able to observe and manage their resources; for example, they could adapt themselves to changes in client traffic without compromising their desired level of performance while respecting a cost budget. In this thesis we refer to applications with this kind of abilities as *resource-aware*. *Resource-aware applications* pose interesting challenges for software development methods. For resource-aware applications, the need to include deployment decisions in the design phase is even more crucial because resource management is part of the application logic.

## 1.2 Research Questions

As discussed above, new business models in computing technology are appearing on the market. Consequently, new challenges arise in software development methods, such as the need to express and compare deployment decisions in the design phase. These challenges make it natural to integrate new characteristics such as resource-awareness in software applications to keep desired levels of performance.

Recent literature on software architectures defines *deployment* as various activities that place a given software application on computing devices. Deployment happens after the application has been designed, implemented, validated, and is ready for operation [54, 66]. Software architects, who want to make appropriate final deployment decisions, need an effective deployment model with the following elements [66]: (1) software system elements, their configuration and their parameters; (2) hardware system elements, their configuration and their parameters; (3) any constraints on the system's elements and/or their parameters; and (4) formal definitions of the quality of service dimensions. It would be useful to have these elements described in the design phase, so that deployment decisions could be included early in the software development process and the managements of resources could be integrated as part of the global design of the application. The overall goal of this thesis is

> *to develop a methodology to express and compare different deployment decisions for resource-aware distributed applications during the design phase.*

To achieve this goal, this thesis will address the following specific research questions:

***RQ1:*** How can we model deployment architectures for resource-aware distributed applications?

***RQ2:*** How can we express architectural decisions related to quality of service in models of resource-aware distributed applications?

***RQ3:*** How can we estimate and quantify quality of service in models of resource-aware distributed applications?

***RQ4:*** How can we compare the suitability of deployment architectures in models of resource-aware distributed applications?

This thesis consists in a number of publications which describe research results achieved in the direction of answering these questions. Moreover, the methodology we are looking for should meet the following requirements:

***Concurrent and distributed object-oriented framework.*** Object orientation is the most popular programming and design paradigm. In fact, it is the leading approach used in the software industry today. Object orientation provides good structuring mechanisms and flexibility in applying changes to artifacts produced during the software development process. The combination of object orientation and concurrency is useful because it can increase responsiveness and throughput of distributed applications. For the methodology presented in this thesis, it is desirable to support concurrency and object orientation, in order to be close to the *mindset* used in commercial and industrial software development.

***Model-based approach.*** A model is mainly a representation of some aspects of interest from a real-world application. Thus models contain less complexity than real-world applications [13]. By decreasing the amount of complexity, we may be able to more easily understand those aspects of interest, and to analyze specific properties related to them. Therefore it is desirable for the methodology presented in this thesis to be model-based.

***Formality.*** Formal methods are important because they seek to increase the reliability of systems by adding rigor to the design process [17]. For the methodology presented in this thesis, it is desirable to have a design notation that offers precision, unambiguity, and abstractions based on a formal syntax and a well defined semantics.

## 1.3 Structure of this Thesis

The rest of this thesis is structured as follows: Chapter 2 introduces the ABS language and discusses its underlying concepts. ABS is a formal modeling language based on a concurrent and distributed object-oriented framework. The methodology presented in this thesis uses the ABS language. Chapter 3 gives a short summary of each of

the papers collected in this thesis. Chapter 4 concludes the thesis by answering the research questions presented in this chapter and suggests some directions for future research. Part II collects the full content of the research papers in the thesis.

CHAPTER 2

# The ABS Language and its Underlying Concepts

ABS [34, 35, 41] is an abstract behavioral specification language with a formal semantics and a Java-like syntax. ABS targets the executable modeling of concurrent, distributed, and object-oriented systems. ABS adapts the concurrent model of the *Creol* language [42, 45] and was developed and used by the European Project HATS [35] (Highly Adaptable and Trustworthy Software using Formal Models). The modeling language features developed in this thesis are presented as extensions of ABS.

In this chapter we discuss some basic concepts of concurrent object-oriented modeling languages and then we position ABS in this context. For this purpose, Sections 2.1–2.3 introduce concurrent languages, object-oriented languages and formal modeling languages, respectively. Section 2.4 introduces ABS specifically and Section 2.5 explains how ABS fits into the context of HATS.

## 2.1 Concurrent Languages

*Concurrent programs* are present in a wide range of software applications. *Concurrency* is a property of systems in which several actions may be executed at the same time. Concurrent programs are inherently more complex than sequential programs; for example, *race conditions* can cause undesired behaviors in these kinds of programs. In concurrent programs, the goal is to have a number of processes (with multiple threads of control), where each process is a sequential program, working and cooperating together in a meaningful manner by communicating and synchronizing with each other [8].

*Communication* is programmed using *shared variables* or *message passing*, depend-

ing on the underlying architecture. In a shared memory setting, processes interact through shared variables. In this setting, processes can concurrently access shared variables, which may generate race conditions because the result of the execution can depend on the speed or scheduling of the different processes; e.g., one process writes to a variable that a second process reads, but the first process continues its execution and manages to change the value of the variable again before the second process sees the results of the first change. In a distributed memory setting with networks of machines, the interaction mechanism is message passing, where one process sends a message that is received by another process [8].

Independent of the form of communication, processes need to synchronize. Correct synchronization coordinates concurrent activities so they can run efficiently, consistently and predictably. We also need to take into consideration that too much coordination reduces concurrency and too little leads to undesired behaviors [71].

Two different kinds of process *synchronization* mechanisms are *mutual exclusion* and *condition synchronization*. Mutual exclusion ensures that statements in different processes do not execute at the same time. *Critical sections* contain shared components and use mutual exclusion to protect these shared components. As examples of *shared components* we have variables, objects, etc. Condition synchronization occurs whenever one process needs to wait for another. In this case, condition synchronization delays processes until certain conditions are true [8]. Mutual exclusion and condition synchronization may be implemented using locks, semaphores, or monitors [8].

Processes in distributed programs usually execute on different processors. In a distributed program, communication channels are typically the shared component. In this case processes must communicate using the channels in order to interact, and the main concern is interprocess communication. The main interaction models for distributed programs are remote procedure calls (RPC), rendezvous, and message passing [8].

RPC and rendezvous are operations with a two-way communication flow. In these operations the thread of control is transferred with the call (from the caller to the callee and then back to the caller), and the caller activity is *blocked* until the return values from the call have been received. Conceptually RPC creates a new process for handle each call. In contrast rendezvous uses an existing process to handle the calls (this is sometimes called extended rendezvous) [8].

In contrast to RPC and rendezvous, message passing uses a one-way communication flow and does not transfer control between the communicating parties. Message passing can be *synchronous* and *asynchronous*. When message passing is synchronous, the sender and the receiver must both be ready to transfer the message, and the sender will not continue its execution until the receiver has received the message. A call in this case can be captured by first sending an invocation message and then receiving a reply message, where the calling process is blocked between the two synchronized

messages (this is sometimes called simple rendezvous). For distributed systems, this form of synchronization may easily result in unnecessary delays due to the blocking of internal activity in the processes, when for example, the sender waits for the receiver to be ready for communication [8, 40].

When message passing is asynchronous, the emission of messages is always possible, regardless of when the receiver accepts the message. In this case the sender can continue its execution and does not need to block [8]. This approach is well-known from the Actor model [2]. However, basic actors do not distinguish between reply messages and invocation messages [40], so two-way communication must be encoded if needed.

## 2.2 Object-Oriented Languages and Concurrency

Object orientation is an important paradigm that is combined with concurrency in programming languages such as Java and C♯. In particular for distributed systems, object orientation is claimed to be one of the best programming techniques, superior to other programming styles [18]. In the object-oriented programming style, a system is described as a collections of objects [7]. The concept of objects was introduced by Ole-Johan Dahl and Kristen Nygaard in Simula 67, a language for both process description and programming with discrete event simulation [27, 28].

The major concepts and principles of object orientation can be summarized as follows [7, 18, 28, 59, 71]: *Objects* are self-contained units that have their own local data and operations acting on these data. Objects are run-time instances of classes and communicate by sending messages to each other. *Classes* with *attributes* (local data) and *methods* (operations) are the main units for describing and structuring programs. Classes act as program patterns that provide initial values to the attributes and implement the behavior of the methods. *Messages* are requests from a sender to a receiver to execute a method. *Interfaces* expose the behaviors implemented in the methods, so that objects can send requests to execute them. Classes implement one or more interfaces. *Encapsulation* is a way of *hiding* and *restricting* the direct access to the data contained in the attributes of classes and objects. *Subtype polymorphism* and *dynamic dispatch* allow interfaces to be implemented by different classes. Finally, *inheritance* allows a class to be defined as an extension of another by *reusing* existing implementations, leading to more concise programs. An object-oriented language includes most of all of these features, but there are variants. For example, classes may be replaced by object cloning and inheritance by delegation (e.g, Self [22]).

Considering the initial idea of object orientation introduced in Simula 67, it is natural to think that concurrency could be introduced by letting objects execute their actions in parallel. In this context, the objects would act as concurrent processes, and distribution would be obtained when objects run in parallel and interact by re-

mote method calls. However this is not how most programming languages integrate concurrency and object orientation [7, 18, 71]. Processes and objects are in principle independent of each other and interact when processes invoke methods contained in objects [71]. For example, both objects and threads can be dynamically created in Java, and objects in Java are shared components in the sense of Section 2.1 and can be accessed concurrently by separate threads. Consequently race condition problems emerge when, for example, two threads executing methods in the same object, write non-deterministically in the same variables. Unless concurrency, synchronization and communication are carefully integrated in the language, a concurrent and distributed object-oriented language can be difficult to use.

Discussions about how languages integrate concurrency and object orientation can be found in [7, 68, 71]. Based on these references, the rest of this section presents different ways of combining concurrency and object orientation. The topics and the presentation follow a similar structure as the one presented by Wyatt *et al.* [71].

***Process management.*** Concurrent object-oriented languages use different approaches for creating, activating and destroying processes.

- *Process creation.* Most concurrent object-oriented languages use one of two approaches to start multiple processes or threads. In the first approach, programmers use special mechanisms for spawning threads. For example, Java has a class `Thread` in the package `java.lang`. Programmers use instances of this class to explicitly create new threads in Java. In this approach, it is the responsibility of the programmer to explicitly implement and control the concurrent activity. Programmers use low-level synchronization mechanisms such as synchronized blocks, locks, monitors and semaphores to implement mutual exclusion and condition synchronization [8].

  In the second approach, threads may be created automatically at message reception, like in the Actor model [2]. In this case the threads are encapsulated within objects. In this approach, the semantics of the language is the one creating and controlling the concurrent activity and guaranteeing mutual exclusion.

- *Process termination.* Processes may be terminated explicitly or implicitly after a message has been processed. When process termination is implicit, processes terminate after replying to a message. This is similar to the concept of RPC explained in Section 2.1. When process termination is explicit, programmers need to explicitly decide when to terminate the processes, otherwise the processes continue to execute after replying to messages and are available to respond to other messages.

- *Process activation.* Processes may be activated when they are created, or remain suspended until they receive a message. The first method causes more paral-

lelism because it lets processes run without messages. Most object-oriented languages wait for messages to arrive before they activate processes. However some languages have processes which are automatically activated; e.g., the run method in active object languages such as Creol [40, 42, 45].

***Communication features.*** In concurrent object-oriented languages, objects communicate by sending messages to each other.

- *Synchronization.* Communication in object-oriented languages is either synchronous or asynchronous.

  As explained in Section 2.1, synchronous communication requires the sender and the receiver to agree on the message transmission.

  Asynchronous communication eliminates this synchronization and can increase concurrent activity. With asynchronous communication, processes can continue executing without waiting for an answer to their messages. Here if a reply is needed, it must be programmed explicitly.

  In eager invocation, asynchronous communication [42] is extended with a "*mailbox*" using future variables [29]. In this case the sender continues executing and the future variable or mailbox acts as a placeholder for the result. The sender executes until it needs to access the future variable. If the result has been returned at this point, the sender continues, if not it blocks and waits for the result. Futures decrease the waiting for a reply and thereby increase concurrency.

- *Message acceptance and message processing.* Objects receive and process messages implicitly or explicitly. In the implicit case, messages are accepted automatically. Some priority can be added to the messages if needed. Objects may process messages in the order they are received, non-deterministically, or by the priority assigned to them. In the explicit case, the objects control when they receive and how they process messages.

***Inheritance.*** A class can inherit methods and attributes from the classes in a class hierarchy, which facilitates code reuse among classes. In this case subclasses often behave as specialized versions of their parents.

Approaches to inheritance vary from fully static to fully dynamic. In the dynamic case, the run-time system determines the appropriate method definition to activate (so-called late binding or dynamic dispatch). In a concurrent object-oriented environment, this late binding may be a problem because there is a potential for conflicts among inherited methods and synchronization requirements [31, 32, 57].

> In the static case, the code of the method is copied at compilation time. This is simple and efficient but wastes run-time memory by replicating code. Additionally there are combined approaches that try to compensate for the problems and disadvantages of the purely dynamic or static approaches.

We have seen that concurrency can be combined with object orientation in several ways, depending on whether processes and objects are seen as distinct concepts or unified into so-called concurrent objects. Furthermore, we have seen that there are choices with respect of how processes interact and synchronize.

## 2.3 Formal Modeling Languages and Concurrency

Models are widely used in software development mainly to understand existing systems that need to be replaced or redesigned and improved, or to design new systems. Magee & Kramer [53] define a model as a simplified representation of the real world and, as such, a model only includes those aspects of the real-world system that are relevant to the problem at hand. When modeling concurrent and distributed systems, the idea is to understand and design the desired behavior of real concurrent and distributed programs. So the models abstract much of the details of the actual implementation of these programs and typically focus on the synchronization aspects of the concurrent behavior.

Formalisms for modeling and understanding concurrent systems constitute an active research field, and new formalisms are frequently proposed and tested. The reason for introducing new formalisms is to increase expressiveness in the sense of making a concurrent model easy to understand and to show that it behaves correctly; i.e., new formalisms aim at reducing complexity related to concurrent behavior in the models. These formalisms include process algebras [37, 53, 58], Petri nets [39], labelled transition systems [11], Actor models [2], distributed objects [21, 42], action systems [10, 60], rewriting logic [55], etc.

In the field of process algebra and labelled transition systems, researchers have built tools that support automatic verification techniques using, for example, bisimulation checking, refinement and model checking. These tools can guarantee the correctness of models with finite state, but it is hard to model control flow that depends on data and these approaches are challenged by the state space explosion generated when composing various processes. These limitations make these formalisms hard to apply to real systems.

When modeling is combined with the object-oriented approach, it is possible to come closer to the initial idea of objects proposed by Dahl and Nygaard [27, 59] where objects simulate interacting real-world entities. In this context, objects are

independent units interacting by means of method calls [21, 42, 45]. With respect to the analysis, distributed objects combined with the characteristics of Actors make it possible to use techniques for compositional reasoning [30] on more complex models closer to real-world systems.

The formalisms presented above focus on the description of functional requirements of systems ignoring non-functional requirements such as performance, which depends on time and on the underlying infrastructure, as discussed in Chapter 1. In response to time-dependent requirements, there have been efforts dedicated to real-time specifications, including extensions of the formalisms listed above, such as timed automata [52] and linear hybrid automata [36] which also generate state space explosion. However there is not much research addressing non-functional requirements involving more specific descriptions of underlying infrastructures, here as an example in this direction, Verhoef *et al.* [67] shows an extension of VDM++ for the deployment of embedded real-time systems.

## 2.4 The ABS Language

ABS is an acronym for Abstract Behavioral Specification. ABS is a modeling language for the development of executable distributed object-oriented models. The kernel of the language is called Core ABS [41]. The main characteristics of Core ABS can be listed as follows [34, 35, 41]: (1) it has a formal syntax and semantics, (2) it has a clean integration of concurrency and object orientation based on concurrent object groups (COGs) [41, 63], and permits synchronous as well as asynchronous communication [29, 42] akin to Actors [2] and Erlang processes [9], (3) it offers a wide variety of complementary modeling alternatives in a concurrent and object-oriented framework that integrates algebraic datatypes, functional programming and imperative programming, (4) compared to object-oriented programming languages, it abstracts from low-level implementation choices such as data structures, and (5) compared to design-oriented languages like UML diagrams, it models data-sensitive control flow and it is executable. ABS is supported by code generators into, e.g., Java, Scala, and Maude [26]. For the work in this thesis, the Maude back-end of ABS has been extended to experiment with the language extensions and to perform simulations of examples.

In addition, ABS supports the modeling of variability in software product line engineering [23, 25, 61, 62] through delta-oriented specifications and feature models. Unlike most object-oriented languages, ABS does not support class inheritance and method overloading, instead code reuse is achieved by applying delta-oriented programming techniques [24, 62]. However, as delta-oriented modeling techniques has not directly been used in this thesis, this topic will not be further discussed.

ABS also provides explicit and implicit time-dependent behavior [15] and the mod-

eling of deployment variability [6, 43, 44, 47, 48, 51]. These extensions are part of the contributions of this thesis, consequently they are covered in detail in the papers collected in Part II of the thesis. The rest of this section will focus on the layered architecture of Core ABS.

### 2.4.1 The Functional and Imperative Layers of Core ABS

Core ABS combines a functional and an imperative layer [34, 35, 41].

***The functional layer.*** The functional layer of Core ABS is used to model computations on the internal data of the imperative layer. It allows modelers to abstract from implementation details of imperative data structures at an early stage in the software design and thus allows data manipulation without committing to a low-level implementation choice. The functional layer combines a simple language for parametric algebraic data types (ADTs) and a pure first-order functional language with definitions by case distinction and pattern matching. Core ABS includes a library with four predefined datatypes, Bool, Int, String and Unit, and parametric datatypes such as lists, sets, maps, etc. The predefined datatypes come with arithmetic and comparison operators, and the parametric datatypes have built-in standard functions. The type Unit is used as a return type for methods without explicit return value. All other types and functions are user-defined.

***The imperative layer.*** The imperative layer of Core ABS allows modelers to express cooperation between concurrent objects through communication and synchronization. The imperative layer combines a simple imperative language, concurrent object groups (COGs) [41, 63] and asynchronous communication [29, 42].

The *concurrency model* of Core ABS consists of a two-tiered model where the upper tier is based on the Actor model [2] with asynchronous communication [42] and no shared state, while the lower tier uses concurrent object groups (COGs) and cooperative scheduling with synchronous and asynchronous communication. Using COGs, a program's components are represented as objects or as groups of objects, and the behavior of the overall system results from the collaboration and interaction between the COGs. In Core ABS, COGs communicate and synchronize using asynchronous method calls and futures [29]. By using futures, it is always possible for an object to call a method from another object in a different COG, continue with its execution and then resynchronize later on the result, whenever it is needed. As discussed in Section 2.2, this synchronization technique reduces the waiting for replies and increases concurrency. In addition, asynchronous communication between COGs contributes to represent the interaction in a natural manner and closer to real-world systems. In the lower tier of Core ABS, synchronous method calls within a COG represent sequential

execution of code, which means that the caller is blocked and execution continues in the callee until control is returned. Asynchronous method calls within a COG create a new process that executes the code of the callee method, while the execution of the caller continues. Multitasking in COGs is not preemptive (decided by a scheduler). On the contrary, it is explicitly decided in the model when control can be transferred to another process (so-called cooperative scheduling). In ABS, a process may suspend, allowing another pending process to be activated. However, the suspending process does not signal a particular other process, instead the selection of the new process is left to the scheduler. In between the explicit scheduling points, only one process is active inside a COG, which means that race conditions are avoided.

### *2.4.2 Design Choices of Core ABS*

Based on the discussion of choices for language design in Section 2.2, we can summarize the design choices of Core ABS as follows:

***Process management.*** Processes in Core ABS are encapsulated within COGs. The semantics of Core ABS guarantees mutual exclusion. Processes are created automatically at method call reception and they terminate after they finish the execution of the method call. Core ABS combines active and reactive behavior of objects, which means that if an object has a `run` method, it is automatically activated. Inside COGs, process may suspend, allowing other pending processes to be activated.

***Communication features.*** Core ABS allows synchronous and asynchronous communication. Synchronous communication within a COG represents sequential execution of code. Asynchronous communication within a COG creates a new process that executes the code of the callee method. Asynchronous communication between COGs is based on the Actor model [2] and uses futures [29]. Method calls in Core ABS are accepted automatically. The cooperative scheduler inside COGs non-deterministically decides the order in which method calls are processed.

***Inheritance.*** Core ABS does not support class inheritance and method overloading.

***Note:*** The details of the sequential execution of code inside a COG is orthogonal to the methodology presented in this thesis. Although the simulation tool, implementing most of the proposed extensions of this methodology, supports groups of objects, the papers in this thesis focus on single object COGs (i.e., concurrent objects) to simplify the presentation.

## 2.5 The HATS Project

HATS stands for Highly Adaptable and Trustworthy Software using Formal Models and was a project supported by the EU's 7th Framework Program. The main idea behind the HATS project was to develop a formal method for the design, analysis, and implementation of highly adaptable software systems that are characterized by a high degree of trustworthiness [35]. As the technical core for supporting this formal methodology, HATS developed and used ABS. ABS is not only a modeling notation, it also has a tool suite that helps to automate the software engineering process. An overview of the tool suite is given in [69].

One of the challenges in the HATS project was to make ABS well-suited for the formal modeling of variability in software product lines. One dimension of this challenge was a *configurable deployment architecture* which in part inspired the work presented in this thesis. The methodology that HATS proposed includes an integration of low-level concepts such as system time, resource-restrictions, latency, and scheduling at the level of the ABS language. This integration is related to the contributions of this thesis and will be presented in the rest of this thesis.

CHAPTER 3

# Overview of the Research Papers

The research contributions of this thesis are presented in four papers which are briefly summarized in this chapter. The full content of the papers appears in Part II, as in their original publications, but have been reformatted to fit the structure of this thesis. The language that is used and extended in the papers is Core ABS. Paper 1 extends Core ABS with real-time in order to express explicit time dependent behavior using a dense time domain (resulting in the language Real-Time ABS), and with application-level scheduling policies. Paper 2 integrates deployment architectures for CPU resources and Real-Time ABS. Paper 3 presents a case study that models virtualized services and distributed applications using Real-Time ABS and its integration with deployments architectures. Paper 4 combines deployment architectures, cost estimation based on worst-case cost analysis for memory usage and Core ABS. Additionally, Section 3.5 lists other publications, to which the author of this thesis has contributed during her Ph.D. studies, and which are not presented as part of this thesis, but are related to the overall goal presented in Section 1.2.

## 3.1 Paper 1: User-defined Schedulers for Real-time Concurrent Objects

***Authors:*** Joakim Bjørk, Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte and Silvia Lizeth Tapia Tarifa.

***Publication:*** Journal of Innovations in Systems and Software Engineering, 2013 [15].

***Summary:*** This paper presents an approach to express and analyze application-specific scheduling decisions during the software design phase. For this purpose, the paper introduces Real-Time ABS, a high-level modeling language for real-time con-

current objects with user-defined schedulers and deadlines for method calls.

Real-Time ABS extends the syntax and semantics of Core ABS with *time-dependent behavior*. In this paper, the Real-Time ABS extension allows models to incorporate explicit manipulation of time from a dense time domain, which may be used to represent execution time inside methods. The local passage of time is expressed in terms of durations (as in, e.g., UPPAAL [52]), but in Real-Time ABS durations may either suspend a process or block an object for a certain amount of time.

Real-Time ABS also extends the syntax and semantics of Core ABS with *deadlines for method calls* and with the ability to refine the non-deterministic per-object scheduler by allowing each concurrent object to embody its own *user-defined scheduling policy* for its process queue. To be able to express user-scheduling policies, we lift run-time processes to the level of the modeling language.

By expressing per-object scheduling policies at the software *modeling* level, we may analyze and resolve deployment requirements related to performance at design time. For example, we may use deadlines associated to method calls to measure and compare the performance of the same model using different user-defined scheduling policies such as, first in first out, earliest deadline first, shortest job first, conditional scheduling depending on the size of the process queue, etc.

This paper also reports results obtained from a tool that implements Real-Time ABS extending the Maude interpreter of Core ABS. This tool can be used for simulation and measurements of Real-Time ABS models. Finally this paper presents a series of examples showing how to express user-defined schedulers in Real-Time ABS and results from running the examples in the tool.

## *3.2 Paper 2: Integrating Deployment Architectures and Resource Consumption in Timed Object-Oriented Models*

***Authors:*** Einar Broch Johnsen, Rudolf Schlatte and Silvia Lizeth Tapia Tarifa.

***Publication:*** Research Report 438, Department of Informatics, University of Oslo, February 2014. Submitted to the Journal of Logic and Algebraic Programming, in second round revision [51]. This work improves and unifies results from three conference papers [43, 44, 48].

***Summary:*** This paper presents a complete formal syntax and semantics that integrates Real-Time ABS with (1) primitives for expressing deployment infrastructures, (2) primitives for resource management, and (3) the ability to express user-defined

execution costs that capture resource consumption of executable models. The underlying deployment infrastructure of the targeted system is part of the system model, but defaults are provided which allow the modeler to ignore the deployment aspects of the model if desired.

In this paper, the syntax and semantics of Real-Time ABS allows explicit and implicit manipulation of time. In addition to the explicit manipulation of time explained in Paper 1, the execution time can also be expressed implicitly. In this case time can be observed by measurements of the executing model. In this way, executable models could for example, compare time values or monitor a system's response time.

To model deployment infrastructures, we introduce the concept of *deployment components* [44]. A deployment component captures a location in a deployment architecture and is a resource-restricted execution context for a set of concurrent object groups (COGs) focusing on processing resources. A deployment component also controls how much computation can occur in this set of COGs between observable points in time. Deployment components may be dynamically created and are parametric in the amount of processing resources they provide to their objects. To simplify the presentation, this paper focuses on single object COGs.

To express *resource management*, we introduce primitives that allow an object to observe the execution capacity of its current deployment component and the average load of its deployment component over time. With this information, objects can detect and predict poor performance. We also introduce primitives that allow objects to request additional execution capacity from other deployment components and to migrate between deployment components.

To model resource consumption, we introduce *user-defined costs*. In our semantics, each computation step has an associated cost which is specified by a user-defined cost expression or by a default value.

The proposed approach of this paper, that uses these extensions, can be characterized by having a separation of concerns between *execution cost*, expressed in the model, and *execution capacity*, expressed in the deployment component. This separation of concerns makes it easy to compare timing and performance for different deployments of a system already during the design phase (rather than to assume that this relationship is fixed in terms of, e.g., specified execution times).

The explicit representation of deployment architectures and resource consumption allows application-level measurement of response time and the modeling of resource management using *load balancing strategies* expressed in software models in a very natural and flexible way. In addition, performance analysis may be applied by comparing deployment possibilities that vary in, for example, the deployment infrastructure, the execution capacity, the client traffic, the resource management strategy, etc.

The semantics presented in this paper has been used to extend the ABS tool suite,

in particular the Maude interpreter for Real-Time ABS. This paper also includes a number of examples with simulation results concerning performance evaluation.

## 3.3 Paper 3: Modeling Resource-Aware Virtualized Applications for the Cloud in Real-Time ABS

***Authors:*** Einar Broch Johnsen, Rudolf Schlatte and Silvia Lizeth Tapia Tarifa.

***Publication:*** Formal Methods and Software Engineering. Proceedings of the 14th International Conference on Formal Engineering Methods, ICFEM 2012 [50]

***Summary:*** This paper shows how our approach with deployment components in Real-Time ABS [51] (presented in Paper 2) may be used to model virtualized systems in a cloud environment. The models consist of a cloud provider which offers billable services to a client application. The client application is a scientific application that concurrently processes and merges heavy graphics for scientific purposes.

The model of the cloud provider is composed of dynamically created virtual machines which are modeled using deployment components with given CPU capacities. The communication interface of the cloud provider allows a client application to create machines with a desired execution capacity, acquire machines to start task executions, release machines and finally get the accumulated usage cost.

The client application is a model of the Montage toolkit [38]. Montage is a portable software toolkit for generating science-grade mosaics by composing multiple astronomical images. Montage is a good candidate for cloud deployment because it manipulates a high volume of data and because it can be configured to have a partially ordered workflow with highly parallelizable tasks.

We analyze the behavior of the whole system by means of simulations using the extended ABS tool suite. We compare and validate our results with previous results obtained for the Montage toolkit using a specialized simulation tool. We also show new results comparing different machine allocations strategies at the client application level.

## 3.4 Paper 4: Simulating Concurrent Behaviors with Worst-Case Cost Bounds

***Authors:*** Elvira Albert, Samir Genaim, Miguel Gómez-Zamalloa, Einar Broch Johnsen, Rudolf Schlatte and Silvia Lizeth Tapia Tarifa

***Publication:*** Proceedings of the 17th International Symposium on Formal Methods, FM 2011 [6]

***Summary:*** This paper presents the formal syntax and semantics of an integration that shows how Core ABS extended with deployment components can be combined with symbolic upper bounds for estimating memory consumption.

As explained in Paper 2, a deployment component imposes resource-restrictions on the concurrent execution of objects deployed on it. The main idea of the approach proposed in this paper is to apply *static worst-case cost analysis* to the sequential parts of a model and as a result to get estimations of resource consumption, for which practical cost analysis methods exist, while in the concurrent part of the model to use simulations measuring the resource consumption inside deployment components. How resources are assigned and consumed inside a deployment component depends on a *cost model* which abstracts from the allocation and deallocation of resources. In this paper we propose a generic cost model and integrate it in the semantics of the language. Following this idea, the working example consider a cost model for memory consumption as an instance of the generic approach.

This paper also reports results obtained from a prototype tool that implements the semantics presented in this paper and from the COSTABS tool [3] that automatically calculates the upper cost bounds for sequential code written in ABS. These tools can be used for simulations and measurements. This paper also presents results from applying these tools to the example of the paper.

## 3.5 Additional Publications

This section lists other publications to which the author of this thesis has contributed during her Ph.D. research, but which are not directly included as part of this thesis. These papers are related to the goal of this thesis or correspond to shorter and/or preliminary versions of the work reported in this thesis.

- ***Title:*** Formal Modeling and Analysis of Resource Management for Cloud Architectures: An Industrial Case Study Using Real-Time ABS.

***Authors:*** Elvira Albert, Frank S. de Boer, Reiner Hähnle, Einar Broch Johnsen, Rudolf Schlatte, Silvia Lizeth Tapia Tarifa and Peter Y. H. Wong

***Publication:*** Journal of Service Oriented Computing and Applications [5]

- ***Title:*** A Formal Model of Object Mobility in Resource-Restricted Deployment Scenarios.

  ***Authors:*** Einar Broch Johnsen, Rudolf Schlatte and Silvia Lizeth Tapia Tarifa

  ***Publication:*** Proceedings of the 8th International Symposium on Formal Aspects of Component Software, FACS 2011 [48]

- ***Title:*** Modeling Application-Level Management of Virtualized Resources in ABS.

  ***Authors:*** Einar Broch Johnsen, Rudolf Schlatte and Silvia Lizeth Tapia Tarifa

  ***Publication:*** Formal Methods for Components and Objects, 10th International Symposium, FMCO 2011 [47]

- ***Title:*** A Formal Model of User-Defined Resources in Resource-Restricted Deployment Scenarios.

  ***Authors:*** Einar Broch Johnsen, Rudolf Schlatte and Silvia Lizeth Tapia Tarifa

  ***Publication:*** Proceedings of the 2nd International Conference on Formal Verification of Object Oriented Software, FoVeOOS 2011 [49]

- ***Title:*** Integrating Aspects of Software Deployment in High-Level Executable Models.

  ***Authors:*** Einar Broch Johnsen, Rudolf Schlatte and Silvia Lizeth Tapia Tarifa

  ***Publication:*** Proceedings for the Norsk Informatikk Konferanse, NIK 2011 [46]

- ***Title:*** Validating Timed Models of Deployment Components with Parametric Concurrency.

  ***Authors:*** Einar Broch Johnsen, Olaf Owe, Rudolf Schlatte and Silvia Lizeth Tapia Tarifa

  ***Publication:*** Proceedings of the 1st International Conference on Formal Verification of Object-Oriented Software, FoVeOOS 2010 [44]

- ***Title:*** Dynamic Resource Reallocation between Deployment Components.

  ***Authors:*** Einar Broch Johnsen, Olaf Owe, Rudolf Schlatte and Silvia Lizeth Tapia Tarifa

  ***Publication:*** Proceedings of the 12th International Conference on Formal Engineering Methods, ICFEM 2010 [43]

- ***Title:*** Models of Rate Restricted Communication for Concurrent Objects.

  ***Authors:*** Rudolf Schlatte, Einar Broch Johnsen, Fatemeh Kazemeyni and Silvia Lizeth Tapia Tarifa

  ***Publication:*** Electronic Notes in Theoretical Computer Science, Volume 274, 2011 [64]

CHAPTER 4

# Conclusions

This chapter returns to the research questions formulated in Section 1.2 and discusses them in connection with the contributions of this thesis. This chapter also suggests directions for future work.

## *4.1 Summary of Contributions*

Resource-aware distributed applications in the terminology of this thesis are applications that are able to, e.g., adapt themselves to changes in client traffic by balancing their service performance and cost. It is a hypothesis of this thesis that it is an advantage to make deployment decisions early in the development of applications, and that this is specially important for resource-aware applications. To make model-based deployment decisions for resource-aware distributed applications, a software architect needs to be able to model deployment infrastructures and then express different deployment decisions together with the model of these distributed applications. In such settings, the analysis activity focuses on evaluating which of the deployment scenarios fits better with respect to a desired level of service quality and reasonable costs. In order to make a quantitative evaluation of performance, software architects could, for example, measure the response time of method calls, measure resource usage (e.g., cpu, memory) simulating normal and overloaded client scenarios, etc.

Let us consider some examples of design decisions related to performance and resource-awareness. Resource-aware distributed applications could monitor the state of the resources provided by their infrastructure; with this information, they are able to react better and to control scenarios where there is a risk of reaching unacceptably long response times. These kind of applications could also implement user-defined load balancers which are in charge of recovering from low performance scenarios. Possible resource balancer strategies could be temporary object migration to physical

devices with low workload or on-demand resource provisioning on virtual and elastic infrastructures such as the ones offered by clouds. Both strategies contribute to a more effective usage of resources.

The rest of this section explains how the methodology presented in this thesis and based on the ABS language, addresses design decisions related to performance and resource-awareness by answering the research questions formulated in Section 1.2.

***RQ1: How can we model deployment architectures for resource-aware distributed applications?***

The methodology proposed in this thesis uses *deployment components* to model deployment architectures in ABS. Deployment components and their formal syntax and semantics are explained in detail in Paper 2. Examples of how deployment components are used to express deployment architectures can be found in Papers 2, 3 and 4. Paper 2 and Paper 3 focus on deployment architectures depicting processing resources while Paper 4 integrates deployment components with a proposed cost model for memory resources in the semantics of the language. Particularly, Paper 3 shows an example modeling an elastic and scalable deployment architecture.

***RQ2: How can we express architectural decisions related to quality of service in models of resource-aware distributed applications?***

The methodology proposed in this thesis offers the possibility to express design decisions related to quality of service as follows: Paper 1 explains how user-defined schedulers can be integrated into the syntax and semantics of ABS and used to control process selection inside concurrent objects. Paper 2 explains how resource management primitives integrated into the syntax and semantics of ABS can be used for monitoring and changing the state of deployment components and also consider object migration between deployment components as a way to adjust load. Resource management primitives and user-defined schedulers can be used independently (as shown in the different examples of Papers 1, 2 and 3) or combined together when modeling load-balancing strategies for improving performance. Both extensions make it easier for the modeler to express resource-awareness.

***RQ3: How can we estimate and quantify quality of service in models of resource-aware distributed applications?***

The methodology proposed in this thesis allows the estimation and quantification of quality of service as follows: Deadlines to method calls and cost associated to pieces of code are integrated in the semantics of ABS as explained in Papers 1 and 2. The examples in Paper 1 use this integration together with durations, representing the explicit passage of time inside methods, to measure the

number of missed deadlines under different user-defined schedulers for activating processes inside concurrent objects. Paper 2 uses this integration together with deployment components in different ways, which are demonstrated in three examples. The first example measures the number of successful requests (the method calls accepted by the service) and successful responses (the calls successfully executed without missing the deadline) under two different assumptions about client behavior and with varying amounts of resources in the deployment components. The second example measures the number of successful responses (the call successfully executed without missing the deadline) using different load balancing strategies and with varying amounts of resources in the deployment components. The third example measures the response time of method calls when either applying or not applying a given load balancing strategy. The case study in Paper 3 demonstrates this integration and deployment components to measure resource usage and response time varying the number of virtual devices. Paper 4 uses deployment components with a cost model for memory resources and static cost analysis to measure resource usage varying the amount of resources in the deployment components.

***RQ4: How can we compare the suitability of deployment architectures in models of resource-aware distributed applications?***

By extracting and organizing the different measurements that were recorded during simulations of different deployment scenarios, it is possible to compare how the same functional model could behaves under different deployment choices. In our work, we have opted for presenting these comparisons visually, using graphics to make it easy to compare the different deployment decisions. Paper 1 shows an example that compares how a functional model performs with different user-defined schedulers. Paper 2 shows examples that compare how a functional model performs with different deployment architectures, how a functional model performs with different load balancing strategies and how the performance of a model improves when it administrates its own resources. Paper 3 includes a case study in which the simulation results show how a functional model performs when it is deployed in architectures containing different amounts of virtual machines. It also shows results about how much the usage of these architectures costs (e.g., deployment models with more virtual machines are faster but more expensive). Consequently, it is possible to compare performance versus cost. Finally, Paper 4 shows an example that compares how a functional model performs when varying amounts of resources in its deployment architecture.

By analyzing the measurements obtained from the simulations of models of resource-aware distributed applications, software architects may be able to acquire a better understanding of the trade-offs from the different deployment scenarios and may be able to make better design decisions for these kinds of applications during the de-

sign phase. Software architects may also be able to better predict and control the performance, scalability and cost of the targeted resource-aware application.

The modeling and analysis of design decisions for resource-aware distributed applications is an interesting problem. The methodology proposed in this thesis supports these activities by including a way to express and control application-specific information such as time-sensitive behaviors, user-defined schedulers, deadlines to an application's different services, the resource cost of an application's services, and deployment-specific information for resource usage.

The methodology proposed in this thesis also meets the requirements stated in Section 1.2. We now revisit these requirements and their rationale:

***Concurrent and distributed object-oriented framework.*** Since the methodology proposed in this thesis extends Core ABS, this methodology uses a concurrent and distributed object-oriented framework. We believe that an object-oriented approach can make our methodology easier to use for software developers.

***Model-based approach.*** The methodology proposed in this thesis extends a modeling language with design-time deployment decisions. We believe a model-based approach is specially important for resource-aware applications which dynamically control their own deployment infrastructure.

***Formality.*** The methodology proposed in this thesis extends a formal language and the considered extensions are also formally defined by means of a formal syntax and semantics that supports rigorous and formal analysis.

## 4.2 Future Work

In this section we identify some possible extensions and future research directions of the methodology proposed in this thesis. The collected papers in this thesis focus on processing and memory resources. One possible extension would be to look at other kind of resources such as bandwidth and electric power. It would also be interesting to lift resources to the level of the modeling language in order to support user-defined resources. In this context, resources could be modeled by means of user-defined cost models which specify how resources are both allocated and consumed. Moreover, the integration of multiple kinds of resources in the same deployment architecture could be useful to analyze more complex scenarios.

This thesis has focused on how to express and formalize resource-aware applications deployed on virtualized infrastructures. The analysis activity has mostly been in terms of simulations. Given that our proposed methodology has a formal semantics,

it seems possible that this methodology can be extended to support stronger analysis techniques, such as deductive verification.

Another line of future work, in the direction of formal analysis methods, is to look at the guarantee of contractual obligations related to performance when invoking methods. One possibility would be to associate preconditions and postconditions to interfaces and let these act as contracts [56] specifying aspects of service level agreements (SLAs). As proposed by the EU FP7 project Envisage [4], it would be useful to integrate a way of representing SLAs in the ABS language, so models of virtualized services could include contracts that guarantee desired levels of performance. In this way, the methodology proposed in this thesis could serve as a foundation for formalizing performance aspects of SLAs.

Looking at language design issues, the proposed methodology in this thesis is built on top of the ABS language. ABS has a particular design with respect of how it integrates object orientation and concurrency. It would be interesting to investigate if this methodology could be easily integrated with other object-oriented and concurrent languages which are based on different design choices with respect to concurrency and object orientation, such as thread-based languages.

Another interesting topic would be to investigate how to model and analyze other non-functional requirements related to deployment, for example, security and fault-tolerance. In this case the first steps could include, for example, to find suitable abstractions to represent these requirements at the abstraction level of the modeling language [19].

# Bibliography


[1] Pekka Abrahamsson, Nilay Oza, and Mikko T. Siponen. Agile software development methods: A comparative review. In *Agile Software Development*, pages 31–59. Springer, 2010.

[2] Gul A. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems*. The MIT Press, Cambridge, Mass., 1986.

[3] Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, and Germán Puebla. COSTABS: a cost and termination analyzer for ABS. In Oleg Kiselyov and Simon Thompson, editors, *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation, PEPM 2012, Philadelphia, Pennsylvania, USA, January 23-24, 2012*, pages 151–154. ACM, 2012.

[4] Elvira Albert, Frank de Boer, Reiner Hähnle, Einar Broch Johnsen, and Cosimo Laneve. Engineering Virtualized Services. In *Proceedings of the 2nd Nordic Symposium on Cloud Computing and Internet Technologies (NordiCloud 2013)*, pages 59–63. ACM, 2013.

[5] Elvira Albert, Frank S. de Boer, Reiner Hähnle, Einar Broch Johnsen, Rudolf Schlatte, S. Lizeth Tapia Tarifa, and Peter Y. H. Wong. Formal modeling and analysis of resource management for cloud architectures. an industrial case study using Real-Time ABS. *Journal of Service-Oriented Computing and Applications*, 2013. To appear.

[6] Elvira Albert, Samir Genaim, Miguel Gómez-Zamalloa, Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Simulating concurrent behaviors with worst-case cost bounds. In Michael Butler and Wolfram Schulte, editors, *Proceedings of the 17th International Symposium on Formal Methods (FM 2011)*, volume 6664 of *Lecture Notes in Computer Science*, pages 353–368. Springer, June 2011.

[7] Pierre America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[8] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison Wesley, 2000.

[9] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[10] Ralph-Johan Back and Kaisa Sere. Stepwise refinement of action systems. *Structured Programming*, 12(1):17–30, 1991.

[11] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[12] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering.*, 30(5):295–310, 2004.

[13] Sami Beydeda, Matthias Book, and Volker Gruhn. *Model-Driven Software Development*. Springer, 2005.

[14] Paul Beynon-Davies, Chris Carne, Hugh Mackay, and Douglas Tudhope. Rapid Application Development (RAD): An empirical review. *European Journal of Information Systems*, 8(3):211–223, 1999.

[15] Joakim Bjørk, Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 9(1):29–43, 2013.

[16] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2Nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[17] Howard Bowman and John Derrick. Issues in formal methods (chapter 3). In H. Bowman and J. Derrick, editors, *Formal Methods for Distributed Processing, A Survey of Object-oriented Approaches*, pages 18–35. Cambridge University Press, 2001.

[18] Manfred Broy. Distributed concurrent object-oriented software. In Olaf Owe, Stein Krogdahl, and Tom Lyche, editors, *From Object-Orientation to Formal Methods*, volume 2635 of *Lecture Notes in Computer Science*, pages 83–95. Springer, 2004.

[19] Manfred Broy and Ralf Reussner. Architectural concepts in programming languages. *IEEE Computer*, 43(10):88–91, 2010.

[20] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging IT platforms: Vision, hype, and

reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599 – 616, 2009.

[21] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Object*. Springer, 2005.

[22] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are shared parts of objects: Inheritance and encapsulation in self. *Lisp and Symbolic Computation*, 4(3):207–222, 1991.

[23] Dave Clarke, Nikolay Diakov, Reiner Hähnle, Einar Broch Johnsen, Ina Schaefer, Jan Schäfer, Rudolf Schlatte, and Peter Y. H. Wong. Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In Marco Bernardo and Valérie Issarny, editors, *Proc. 11th Intl. School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM 2011)*, volume 6659 of *Lecture Notes in Computer Science*, pages 417–457. Springer, 2011.

[24] Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. Abstract delta modeling. In Eelco Visser and Jaakko Järvi, editors, *Proc. Ninth International Conference on Generative Programming and Component Engineering, (GPCE'10)*, pages 13–22. ACM, 2010.

[25] Dave Clarke, Radu Muschevici, José Proença, Ina Schaefer, and Rudolf Schlatte. Variability modelling in the ABS language. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proc. 9th Intl. Symposium on Formal Methods for Components and Objects (FMCO'10)*, volume 6957 of *Lecture Notes in Computer Science*, pages 204–224. Springer, 2012.

[26] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[27] Ole-Johan Dahl. The Roots of Object-orientation: The Simula Language. In Manfred Broy and Erns Denert, editors, *Software Pioneers*, pages 78–90. Springer, 2002.

[28] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. Simula 67, common base language. Technical Report S-2, Norsk Regnesentral (Norwegian Computing Center), Oslo, Norway, may 1968.

[29] Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In Rocco de Nicola, editor, *Proc. 16th European Symposium on*

*Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer, March 2007.

[30] Crystal Chang Din, Johan Dovland, Einar Broch Johnsen, and Olaf Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. *Journal of Logic and Algebraic Programming*, 81(3):227–256, 2012.

[31] Johan Dovland, Einar Broch Johnsen, and Olaf Owe. Reasoning about asynchronous method calls and inheritance. In Chunming Rong, editor, *Proceedings of the Norwegian Informatics Conference (NIK'04)*, pages 213–224. Tapir Academic Publisher, November 2004.

[32] Szabolcs Ferenczi. Guarded methods vs. inheritance anomaly: Inheritance anomaly solved by nested guarded method calls. *ACM SIGPLAN Notices*, 30(2):49–58, February 1995.

[33] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, 2nd edition, 2002.

[34] Reiner Hähnle. The abstract behavioral specification language: A tutorial introduction. In *Formal Methods for Components and Objects*, volume 7866 of *Lecture Notes in Computer Science*, pages 1–37. Springer, 2013.

[35] Reiner Hähnle, Michiel Helvensteijn, Einar Broch Johnsen, Michael Lienhardt, Davide Sangiorgi, Ina Schaefer, and Peter Y. H. Wong. Hats abstract behavioral specification: The architectural view. In Bernhard Beckert, Ferruccio Damiani, FrankS. Boer, and MarcelloM. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 7542 of *Lecture Notes in Computer Science*, pages 109–132. Springer, 2013.

[36] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, LICS '96*, pages 278–292. IEEE Computer Society, 1996.

[37] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., 1985.

[38] Joseph C. Jacob, Daniel S. Katz, G. Bruce Berriman, John Good, Anastasia C. Laity, Ewa Deelman, Carl Kesselman, Gurmeet Singh, Mei-Hui Su, Thomas A. Prince, and Roy Williams. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *International Journal of Computational Science and Engineering*, 4(2):73–87, July 2009.

[39] Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer, 2009.

[40] Einar Broch Johnsen, Jasmin Christian Blanchette, Marcel Kyas, and Olaf Owe. Intra-object versus inter-object: Concurrency and reasoning in Creol. In *Proc. 2nd Intl. Workshop on Harnessing Theories for Tool Support in Software (TTSS'08)*, volume 243 of *Electronic Notes in Theoretical Computer Science*, pages 89–103. Elsevier, 2009.

[41] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2011.

[42] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, March 2007.

[43] Einar Broch Johnsen, Olaf Owe, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Dynamic resource reallocation between deployment components. In J. S. Dong and H. Zhu, editors, *Proceedings of the 12th International Conference on Formal Engineering Methods (ICFEM'10)*, volume 6447 of *Lecture Notes in Computer Science*, pages 646–661. Springer, November 2010.

[44] Einar Broch Johnsen, Olaf Owe, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Validating timed models of deployment components with parametric concurrency. In B. Beckert and C. Marché, editors, *Proceedings of the 1st International Conference on Formal Verification of Object-Oriented Software (FoVeOOS'10)*, volume 6528 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2011.

[45] Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, November 2006.

[46] Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Integrating aspects of software deployment in high-level executable models. In *Proceedings of the Norwegian International Conference - Information Technology (NIK'11)*. Tapir Akademisk Forlag, 2011.

[47] Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Modeling application-level management of virtualized resources in ABS. In Bernhard Beckert, Ferruccio Damiani, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects, 10th International Symposium (FMCO2011)*, volume 7542 of *Lecture Notes in Computer Science*, pages 89–108. Springer, 2011.

[48] Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. A formal model of object mobility in resource-restricted deployment scenarios. In Farhad Arbab and Peter Ölveczky, editors, *Proceedings of the 8th International Symposium on Formal Aspects of Component Software (FACS 2011)*, volume 7253 of *Lecture Notes in Computer Science*, pages 187–204. Springer, 2012.

[49] Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. A formal model of user-defined resources in resource-restricted deployment scenarios. In Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov, editors, *Proceedings of the 2nd International Conference on Formal Verification of Object-Oriented Software (FoVeOOS'11)*, volume 7421 of *Lecture Notes in Computer Science*, pages 196–213. Springer, 2012.

[50] Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Modeling resource-aware virtualized applications for the cloud in Real-Time ABS. In Toshiaki Aoki and Kenji Tagushi, editors, *Proceedings of the 14th International Conference on Formal Engineering Methods (ICFEM'12)*, volume 7635 of *Lecture Notes in Computer Science*, pages 71–86. Springer, November 2012.

[51] Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Integrating deployment architectures and resource consumption in timed object-oriented models. Research Report 438, Department of Informatics, University of Oslo, February 2014.

[52] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.

[53] Jeff Magee and Jeff Kramer. *Concurrency: state models & Java programs*. John Wiley & Sons, Inc., 1999.

[54] Sam Malek, Nenad Medvidović, and Marija Mikic-Rakic. An extensible framework for improving a distributed software system's deployment architecture. *IEEE Transactions on Software Engineering*, 38(1):73–100, 2012.

[55] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[56] Bertrand Meyer. Design by contract. the eiffel method. In *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings*, page 446. IEEE Computer Society Press, 1998.

[57] Giuseppe Milicia and Vladimiro Sassone. The inheritance anomaly: ten years after. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC'04)*, pages 1267–1274. ACM Press, 2004.

[58] Robin Milner. *A Calculus of Communicating Systems.* Springer, 1982.

[59] Olaf Owe, Stein Krogdahl, and Tom Lyche. A biography of Ole-Johan Dahl. In Olaf Owe, Stein Krogdahl, and Tom Lyche, editors, *From Object-Orientation to Formal Methods*, volume 2635 of *Lecture Notes in Computer Science*, pages 1–7. Springer, 2004.

[60] Luigia Petre and Kaisa Sere. Coordination among mobile objects. In Paolo Ciancarini and Alexander L. Wolf, editors, *Proceedings of the Third International Conference onCoordination Languages and Models (COORDINATION'99)*, volume 1594 of *Lecture Notes in Computer Science*, pages 227–242. Springer, 1999.

[61] Ina Schaefer. Variability modelling for model-driven development of software product lines. In David Benavides, Don S. Batory, and Paul Grünbacher, editors, *Proc. Fourth International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'10)*, volume 37 of *ICB-Research Report*, pages 85–92. Universität Duisburg-Essen, 2010.

[62] Ina Schaefer and Ferruccio Damiani. Pure delta-oriented programming. In *Proceedings of the Second International Workshop on Feature-Oriented Software Development, FOSD 2010*, pages 49–56. ACM, 2010.

[63] Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *European Conference on Object-Oriented Programming (ECOOP 2010)*, volume 6183 of *Lecture Notes in Computer Science*, pages 275–299. Springer, June 2010.

[64] Rudolf Schlatte, Einar Broch Johnsen, Fatemeh Kazemeyni, and Silvia Lizeth Tapia Tarifa. Models of rate restricted communication for concurrent objects. *Electronic Notes in Theoretical Computer Science*, 274:67–81, 2011.

[65] Ian Sommerville. *Software Engineering (7th Edition).* Pearson Addison Wesley, 2004.

[66] Richard N. Taylor, Nenad Medvidović, and Eric M. Dashofy. *Software Architecture-Foundations, Theory, and Practice.* Wiley, 2010.

[67] Marcel Verhoef, Peter Gorm Larsen, and Jozef Hooman. Modeling and validating distributed embedded real-time systems with VDM++. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *Proceedings of the 14th International Symposium on Formal Methods (FM'06)*, volume 4085 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2006.

[68] Peter Wegner. Design issues for object-based concurrency. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Object-Based Concurrent Computing*, volume 612

of *Lecture Notes in Computer Science*, pages 245–256. Springer Berlin Heidelberg, 1992.

[69] Peter Y. H. Wong, Elvira Albert, Radu Muschevici, José Proença, Jan Schäfer, and Rudolf Schlatte. The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *STTT*, 14(5):567–588, 2012.

[70] Murray Woodside, Greg Franks, and Dorina C. Petriu. The future of software performance engineering. In *2007 Future of Software Engineering*, FOSE '07, pages 171–187. IEEE Computer Society, 2007.

[71] Barbara B. Wyatt, Krishna Kavi, and Steve Hufnagel. Parallelism in object-oriented languages: A survey. *IEEE Software*, 9(6):56–66, 1992.

# Part II

# Research Papers

CHAPTER 5

# Paper 1: User-defined Schedulers for Real-Time Concurrent Objects *

***Authors:*** Joakim Bjørk, Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte and Silvia Lizeth Tapia Tarifa.

***Publication:*** Journal of Innovations in Systems and Software Engineering, Volume 9, Issue 1, pages 29–43. Springer, 2013.

***Abstract:*** Scheduling concerns the allocation of processors to processes, and is traditionally associated with low-level tasks in operating systems and embedded devices. However, modern software applications with soft real-time requirements need to control application-level performance. High-level scheduling control at the application level may complement general purpose OS level scheduling to fine-tune performance of a specific application, by allowing the application to adapt to changes in client traffic on the one hand and to low-level scheduling on the other hand. This paper presents an approach to express and analyze application-specific scheduling decisions during the software design stage. For this purpose, we integrate support for application-level scheduling control in a high-level object-oriented modeling language, Real-Time ABS, in which executable specifications of method calls are given deadlines and real-time computational constraints. In Real-Time ABS, flexible application-specific schedulers may be specified by the user, i.e., developer, at the abstraction level of the high-level modeling language itself and associated with concurrent objects at creation time. Tool support for Real-Time ABS is based on an abstract interpreter that supports simulations and measurements of systems at the design stage.

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

## 5.1 Introduction

The scheduling problem concerns the allocation of available processors to unfinished processes. This is a non-trivial problem as processes in general are dynamically created, have possibly conflicting needs, and their execution time depends on the size of their input. Operating systems usually use heuristics to guess the optimal ordering of their processes, called scheduling policies. Despite many years of research on optimizing scheduling policies at the level of operating systems, scheduling is largely beyond the control of most existing high-level modeling and programming languages, whose purpose is to relieve the software developer from implementation and deployment details by means of suitable abstractions. However, in general-purpose programming, software engineers increasingly need to express and control not only functional correctness but also quality of service in their designs. On the other hand, operating systems by their very nature do not consider the specific requirements of different applications and this can greatly affect application-level performance. For optimal use of both hardware and software resources, we therefore cannot avoid leveraging scheduling and performance related issues from the underlying operating system to the software engineering level, which necessitates timed semantics for general-purpose modeling and programming languages [27]. Today, domain-specific languages for cyber-physical systems with soft (or firm) real-time requirements start to provide cooperative, non-preemptive scheduling with deadlines at the application level [38]. Furthermore, emerging support for reflection in middleware allows applications to dynamically control the creation of virtual processors and to define user-level schedulers [8]. Flexible application-specific schedulers are needed in new application domains with soft real-time requirements, such as multimedia streaming. To fully exploit increasing platform virtualization, a major challenge in software engineering is to find a balance between these two conflicting requirements of abstraction and deployment control.

This paper presents Real-Time ABS, a high-level abstract behavioral specification language for modeling distributed systems, which supports the integration of realizable abstractions of requirements related to deployment, e.g., scheduling and deadlines. We achieve this by modeling the components of the distributed system as loosely interacting concurrent objects. Concurrent objects are similar to the Actor model [1] and Erlang [6] processes in that they represent a *unit of distribution*; i.e., their interaction does not transfer control. Each concurrent object has a queue of method activations (stemming from method calls), among which at most one may be executing at any time. This model of computation is currently attracting interest due to its potential for distribution on multicore platforms, for example in terms of Scala actors [19], concurrent object groups in Java [36], Kelim lightweight threads [39], and concurrent Creol objects in Java [30].

An important aspect of the concurrent object model in Creol [25] and ABS [24]

is that the scheduling of method activations from an object's queue is *cooperative* but *non-deterministic*. The cooperative scheduling is non-preemptive, but a method activation may yield control at explicitly declared points in its execution, leaving the object idle. When the object is idle, scheduling is non-deterministic in the sense that any enabled method activation may resume its execution. In Real-Time ABS, we extend this computation model with real-time and with the ability to refine the non-deterministic per-object scheduler by allowing each concurrent object to embody its own scheduling policy, tailored towards attaining its quality of service. These high-level scheduling policies provide run-time adaptability, which is beyond the state-of-the-art fixed-priority scheduling in operating systems. Thus, we both leverage and generalize scheduling issues from the operating system to the application level. Our approach is based on a two-level scheduling scheme (e.g., [12,35,40]). At the top level, objects on virtual servers decide on the scheduling of requests by encapsulating an application-level scheduling policy. However, these virtual servers may be realized in terms of one or several virtual or physical machines. The bottom level scheduling, which decides on the arbitration between the virtual servers, is not adressed in this paper.

By expressing per object scheduling policies at the software *modeling* level, quality-of-service and deployment requirements may be analyzed and resolved at design time. For example in a real-time setting, we must guarantee a maximum on average response-time (end-to-end deadlines) or a minimum on the level of system throughput. The main technical contributions of this paper are

- a real-time object-oriented modeling language associating schedulers with concurrent objects and deadlines with method calls, and its formal semantics;
- user-defined schedulers expressed at the abstraction level of the modeling language; and
- tool support based on an abstract interpreter.

The formal semantics of Real-Time ABS rigorously defines the real-time behavior of a system of concurrent objects. However, the expressivity of the Real-Time ABS modeling language means that models in Real-Time ABS are out of scope for automata-based model-checking techniques for full state-space exploration (such as, e.g., Uppaal [26]). Based on the formal semantics, we have therefore implemented a prototype abstract interpreter for Real-Time ABS in Maude [14], an executable rewriting logic framework which also supports real-time rewriting logic [31, 32]. Technically, user-defined schedulers are achieved by allowing a degree of reflection in the language, such that the runtime representation of the process queue of objects is lifted into an expression inside the modeling language itself. The abstract interpreter is not bound to any particular hardware or deployment platform; instead, its architecture is directly based on the concepts of Real-Time ABS. The abstract interpreter can

be used for both systematic simulation and measurement of Real-Time ABS models. Systematic simulations allow us to simulate different runs for our executable models, even before creating a detailed implementation of the software. As for measurements, profiling techniques for Real-Time ABS can be used to, e.g., find bottlenecks, data and communication intensity, maximum queue sizes, race conditions, etc.

***Related Work.*** Real-Time ABS is a real-time extension of ABS [24], which simplifies Creol [9, 25] by, e.g., ignoring class inheritance. This journal paper integrates and extends two previous papers on timed concurrent objects by the authors [7, 10], and additionally introduces the concept of user-defined per object schedulers. A discrete time Creol interpreter was proposed in [7], in which the passage of time is indirectly observable by comparing observations of a global clock. In contrast, duration statements and deadlines for method calls were introduced and formalized with a real-time semantics in [10], without a language interpreter. In this paper, we follow the latter approach at the level of the surface syntax, additionally introduce duration guards which complement duration statements, and develop an abstract interpreter for the resulting language.

An encoding of real-time concurrent objects into timed automata was proposed in [10]. This encoding follows the approach of task automata [18], which allows schedulability analysis in Uppaal [26] by means of the Times Tool [3]. Complementing this work, we develop an abstract interpreter based on the formal semantics of real-time concurrent objects as proposed in this paper, by integrating the real-time model of Real-Time Maude [31, 32] with the Maude interpreter for ABS developed in the EU FP7 project HATS: Highly Adaptable and Trustworthy Software using Formal Models. Real-Time ABS provides a much more expressible language than that of, for example, the Times Tool. This additional expressive power is needed for the high-level description of complex data-intensive systems of real-time concurrent objects enhanced with user-defined schedulers, which is proposed in this paper.

Scheduling has been studied extensively in many different research communities, including real-time [13, 16], parallel algorithms [5, 23], operating systems [22], measurement and modeling [20], and job scheduling [15]. These communities have mostly worked independently with somewhat different concerns, and with little cooperation. Traditional approaches to schedulability analysis, especially in operations research, can handle only a restricted range of events, e.g., periodic ones that are generated with fixed inter-arrival times. In this paper we integrate application-level scheduling of non-uniform dynamically created processes with an executable high-level real-time modeling language.

The state of the art in parallel and distributed programming is the multi-threading paradigm, as in Java and the pthread library (usually in C++). Despite its popularity, multi-threading has some generally recognized drawbacks; for example, shared memory access between threads makes it not suitable for programming distributed

| *Syntactic categories.* | *Definitions.* | | |
|---|---|---|---|
| $T$ in Ground Type | $T$ | $::=$ | $B \mid I \mid D \mid D\langle\overline{T}\rangle$ |
| $A$ in Type | $A$ | $::=$ | $N \mid T \mid N\langle\overline{A}\rangle$ |
| $x$ in Variable | $Dd$ | $::=$ | $\mathtt{data}\ D[\langle\overline{A}\rangle] = [\overline{Cons}];$ |
| $e$ in Expression | $Cons$ | $::=$ | $Co[(\overline{A})]$ |
| $b$ in Bool Expression | $F$ | $::=$ | $\mathtt{def}\ A\ fn[\langle\overline{A}\rangle](\overline{A}\ \overline{x}) = e;$ |
| $v$ in Value | $e$ | $::=$ | $b \mid d \mid x \mid v \mid Co[(\overline{e})] \mid fn(\overline{e}) \mid \mathtt{case}\ e\ \{\overline{br}\}$ |
| $d$ in DurationExpr | | | $\mid \mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e \mid \mathtt{this}$ |
| $br$ in Branch | | | $\mid \mathtt{destiny} \mid \mathtt{deadline}$ |
| $p$ in Pattern | $v$ | $::=$ | $Co[(\overline{v})] \mid \mathtt{null}$ |
| | $br$ | $::=$ | $p \Rightarrow e;$ |
| | $p$ | $::=$ | $_ \mid x \mid v \mid Co[(\overline{p})]$ |

Figure 5.1: Syntax for the functional level of Real-Time ABS. Terms $\overline{e}$ and $\overline{x}$ denote possibly empty lists over the corresponding syntactic categories, and square brackets [ ] optional elements.

applications. In addition, as the thread of execution travels across object boundaries (through synchronous method calls), there is no natural borderline to decompose a program for distributed deployment. New programming languages, e.g., Scala and Erlang (as described in [19] and [28], respectively), employ new paradigms that are fundamentally different from and alleviate some of the shortcomings of mainstream languages like Java, but still provide very limited control to influence the underlying scheduling policy. Some amount of controllability is achieved by Real-Time Specification for Java (RTSJ) [17] which allows assigning priorities to Java threads which can be used by a Fixed Priority Scheduler in the JVM. Schoeberl shows how a system-wide fixed priority scheduler can be defined inside Java itself [37]. Erlang has an efficient scheduler which improves the execution performance, but lacks proper language support to influence the scheduling policy. We are not aware of languages which support per-object user-defined schedulers at the abstraction level of the high-level programming or modeling language, as proposed in Real-Time ABS.

## 5.2 Real-Time ABS

Real-Time ABS is a high-level modeling language for real-time concurrent objects with user-defined schedulers. This section presents the integration of real-time and concurrent object-oriented modeling in Real-Time ABS. Section 5.3 presents the integration of local schedulers into the language.

Real-Time ABS consists of a functional level and a concurrent object level. A Real-Time ABS *model* defines interfaces, classes, datatypes, and functions, and a

*main block* to configure the initial state. Objects are dynamically created instances of classes; their attributes are initialized to type-correct default values (e.g., **null** for object references), but may be redefined in an optional method *init*. Datatypes and functions are specified at the functional level, this allows a model to abstract from concrete imperative data structures inside the concurrent objects.

The concurrent object level focuses on the concurrency, communication, and synchronization aspects of a model, given in terms of interacting, concurrent objects. In Real-Time ABS, every concurrent object has its own queue of processes corresponding to the method activations. All objects are executing concurrently and each object executes one process at a time; in the sequel we refer to this process as the active process of the object.

### 5.2.1 The Functional Level of Real-Time ABS

The functional level of Real-Time ABS is used to model data manipulation in an intuitive way, without committing to specific low-level imperative data structures at an early stage in the software design. The functional level defines user-defined parametric datatypes and functions, as shown in Fig. 5.1. The ground types $T$ consist of basic types $B$ such as Bool and Int, as well as names $D$ for datatypes and $I$ for interfaces. In general, a type $A$ may also contain type variables $N$ (i.e., uninterpreted type names [33]). In *datatype declarations* $Dd$, a datatype $D$ has a set of constructors $Cons$, which have a name $Co$ and a list of types $\overline{A}$ for their arguments. *Function declarations* $F$ have a return type $A$, a function name $fn$, a list of parameters $\overline{x}$ of types $\overline{A}$, and a function body $e$. *Expressions* $e$ include Boolean expressions $b$, variables $x$, values $v$, the self-identifier **this**, constructor expressions $Co(\overline{e})$, function expressions $fn(\overline{e})$, and case expressions **case** $e$ $\{\overline{br}\}$. The expression **destiny** refers to the concurrent object level and denotes the future in which the return from the current method activation shall be stored [9]. The expression **deadline** refers to the timed aspect of a model and denotes the relative deadline of the current method activation. *Values* $v$ are constructors applied to values $Co(\overline{v})$ or **null** (omitted from Fig. 5.1 are values of basic types such as String or the rational numbers Rat). *Case expressions* have a list of branches $p \Rightarrow e$, where $p$ is a pattern. Branches are evaluated in the listed order. Patterns include wild cards _, variables $x$, values $v$, and constructor patterns $Co(\overline{p})$. For simplicity, Real-Time ABS does not currently support operator overloading.

**Example 1** (Dense Time in Real-Time ABS)**.** *We consider a dense time model represented by a type* Duration *that ranges over non-negative rational numbers and is used for time intervals, e.g. deadlines and computation costs. To express infinite (or unbounded) durations, there is a term* `InfDuration` *such that for all other durations* $d_1, d_2$*,* $d_1 + d_2$ *is smaller than* `InfDuration`*.*

```
data Duration = Duration(Rat) | InfDuration;
```

*Predicates are defined in the obvious way, shown here are a unary predicate* `isInfinite` *and the binary less-than-or-equal relation* `lte` *on* Durations*.*

```
def Bool isInfinite(Duration d) =
  case d { InfDuration => True; _ => False;};

def Bool lte(Duration d1, Duration d2) =
  case d1 {
    InfDuration => d2 == InfDuration;
    Duration(v1) => case d2 {
      InfDuration => True;
      Duration(v2) => v1 ≤ v2;
    };
};
```

*Two* Duration *values can be added together. Since there is no operator overloading in ABS, we define the addition of durations as a function* `add`*:*

```
def Duration add(Duration d1, Duration d2) =
  case d1 {
    InfDuration => InfDuration;
    Duration(v1) => case d2 {
      InfDuration => InfDuration;
      Duration(v2) => Duration(v1 + v2);
    };
};
```

*The operators* $\leq$ *and* + *are used for comparison and addition in the underlying datatype of rational numbers.*

### 5.2.2 The Concurrent Object Level of Real-Time ABS

The concurrent object level of Real-Time ABS is used to specify concurrency, communication, and synchronization in the system design. The syntax of the concurrent object level is given in Figure 5.2. An interface $IF$ has a name $I$ and method signatures $Sg$. A class $CL$ has a name $C$, interfaces $\overline{I}$ (specifying types for its instances), class parameters and state variables $x$ of type $T$, and methods $M$ (The *attributes* of the class are both its parameters and state variables). A method signature $Sg$ declares the return type $T$ of a method with name $m$ and formal parameters $\overline{x}$ of types $\overline{T}$. $M$ defines a method with signature $Sg$, local variable declarations $\overline{x}$ of types $\overline{T}$, and a statement $s$. Statements may access attributes of the current class, locally defined variables, and the method's formal parameters. A program's main block is a method body $\{\overline{T}\ \overline{x}; s\}$. There are no type variables at the concurrent object level of Real-Time ABS.

| *Syntactic categories.* | *Definitions.* | | |
|---|---|---|---|
| $C, I, m$ in Name | $IF$ | $::=$ | $\texttt{interface}\ I\ \{\ [\overline{Sg}]\ \}$ |
| $g$ in Guard | $CL$ | $::=$ | $[\texttt{[}a\texttt{]}]\ \texttt{class}\ C\ [(\overline{T\ x})]\ [\texttt{implements}\ \overline{I}]\ \{\ [\overline{T\ x};]\ \overline{M}\}$ |
| $s$ in Stmt | $Sg$ | $::=$ | $T\ m\ ([\overline{T\ x}])$ |
| $a$ in Annotation | $M$ | $::=$ | $[\texttt{[}a\texttt{]}]\ Sg\ \{[\overline{T\ x};]\ s\ \}$ |
| | $a$ | $::=$ | $\texttt{Deadline:}\ d \mid \texttt{Cost:}\ d \mid \texttt{Critical:}\ b$ |
| | | | $\mid \texttt{Scheduler}:\ e \mid a, a$ |
| | $g$ | $::=$ | $b \mid x? \mid \texttt{duration}(d, d) \mid g \wedge g$ |
| | $s$ | $::=$ | $s; s \mid \texttt{skip} \mid \texttt{if}\ b\ \{\ s\ \}\ [\texttt{else}\ \{\ s\ \}] \mid \texttt{while}\ b\ \{\ s\ \}$ |
| | | | $\mid \texttt{return}\ e[\texttt{[}a\texttt{]}]\ x = rhs \mid \texttt{suspend} \mid \texttt{await}\ g$ |
| | | | $\mid \texttt{duration}(d, d)$ |
| | $rhs$ | $::=$ | $e \mid \texttt{new}\ C\ (\overline{e}) \mid e.\texttt{get} \mid o!m(\overline{e})$ |

Figure 5.2: Syntax for the concurrent object level of Real-Time ABS.

*Annotations* in Real-Time ABS are used to extend a model with information to control the quantitative properties of the model, while maintaining a separation of concerns with the qualitative behavior of the model. Annotations $a$ are optional and may be associated with class and method declarations, new objects, and method calls as follows. For *class declarations* and *object creation*, the annotation `Scheduler`: $e$ may be used to override a default scheduling policy with a user-defined policy $e$. User-defined scheduling policies are explained in Section 5.3.

For *method declarations*, the annotation `Cost`: $d$ may be used to override a default cost estimate for the method activation. Cost estimates are functions which depend on the arguments to method calls, so $d$ is an expression over the formal parameters $\overline{x}$ of the method and will be evaluated for the actual parameter values to the call. For the purposes of this paper the cost estimate is assumed to be of type Duration, but one could also consider, e.g., memory or other resources. (Observe that cost expressions are estimations (i.e., they are specified as part of the design and not measured at runtime). We do not consider how to find good cost estimations in this paper. However, a companion tool COSTABS may in many cases automatically assist in inferring cost annotations for methods in Real-Time ABS models [2].) The cost provides a heuristics which may be used for the scheduling of method activations. Method declarations without cost annotations get a default of no cost.

For *method calls*, we consider two annotations. The annotation `Deadline`: $d$ is of type Duration and provides a deadline for the method return. The deadline specifies the relative time before which the corresponding method activation should finish execution. The annotation `Critical`: $b$ specifies the perceived level of criticality of the method activation: `True` indicates that the method activation is *hard* and should be given priority [11]. In contrast, missed deadlines for *soft* method activations

are logged but do not influence scheduling. This way, the caller may decide on the deadline and criticality for the call. Calls without annotations get default values: infinite deadline and soft criticality.

*Right hand side expressions rhs* include object creation **new** $C(\overline{e})$, method calls $[o]!m(\overline{e})$ and $[o.]m(\overline{e})$, future dereferencing $x.$**get** and pure expressions $e$. Method calls and future dereferencing are explained below.

*Statements* are standard for sequential composition $s_1; s_2$, and **skip**, **if**, **while**, and **return** constructs. Assignments may be given optional annotations as explained above. The statement **suspend** unconditionally suspends the active process of an object by adding it to its queue from which subsequently an enabled process is taken for execution. In **await** $g$, the guard $g$ controls suspension of the active process and consists of Boolean conditions $b$ and return tests $x?$ (see below). Just like pure expressions $e$, guards $g$ are side-effect free. If $g$ evaluates to false, the executing process is suspended, i.e., added to the object's queue, and another process is taken from the queue for execution. By means of a user-defined scheduling policy enabled processes can be selected for execution from the object's queue.

*Communication* in Real-Time ABS is based on asynchronous method calls, denoted by assignments $f = o!m(\overline{e})$ to future variables $f$. (Local calls are written **this**$!m(\overline{e})$.) After making an asynchronous method call $x := o!m(\overline{e})$, the caller may proceed with its execution without blocking on the method reply. Here $x$ is a future variable, $o$ is an object expression, and $\overline{e}$ are (data value or object) expressions providing actual parameter values for the method invocation. The future variable $x$ refers to a return value which has yet to be computed. There are two operations on future variables, which control synchronization in Real-Time ABS. First, the guard **await** $x?$ suspends the active process unless a return to the call associated with $x$ has arrived, allowing other processes in the object to execute. Second, the return value is retrieved by the expression $x.$**get**, which blocks all execution in the object until the return value is available. In Real-Time ABS, it is the decision of the caller whether to call a method synchronously or asynchronously, and when to synchronize on the return value of a call. Standard usages of asynchronous method calls include the statement sequence $x := o!m(\overline{e});\ v := x.$**get** which encodes a *blocking call*, abbreviated $v := o.m(\overline{e})$ (often referred to as a synchronous call), and the statement sequence $x := o!m(\overline{e});$ **await** $x?;\ v := x.$**get** which encodes a non-blocking, *preemptible call*, abbreviated **await** $v := o.m(\overline{e})$. As usual, if the return value of a call is of no interest, the call may occur directly as a statement $o!m(\overline{e})$.

*Time.* In Real-Time ABS, the local passage of time is *explicitly expressed* using **duration** statements and **duration** guards. The statement **duration**$(b, w)$ expresses the passage of time, given in terms of an interval between the best case $b$ and the worst case $w$ duration (assuming $b \leq w$), and blocks the whole object. The guard **duration**$(b, w)$, when used inside an **await** statement expresses the suspension time of the process in terms of a similar interval, and lets other processes of the object

run in the meantime. Note that time can also pass during synchronization with a method invocation; this can block one process (via **await** $f$?) or the whole object (via $x = f$.**get**). All other statements (normal assignments, **skip** statements, etc.) do not cause time to pass.

Inside a method body, the read-only local variable **deadline** refers to the *remaining permitted execution time* of the current method activation, which is initially given by a deadline annotation at the method call site or by default. We assume that message transmission is instantaneous, so the deadline expresses the time until a reply is received; i.e., it corresponds to an *end-to-end* deadline. If declared, an integer variable `value` may be assigned a value to express the relative importance of the current method activation with respect to other method activations executing in the object [11]. (By default, the method activation is unimportant and has value zero.)

## 5.3 Scheduling Strategies in Real-Time ABS

Scheduling refers to the way processes are assigned to run on the available processors of a runtime system. We here present the general notions of process scheduling, following Buttazzo [11], and relate these to the context of concurrent objects in Real-Time ABS. A processor is assigned to various concurrent processes according to a predefined criterium which is called a *scheduling policy*. The realization of a scheduling policy as an algorithm which, at any time, determines the order in which processes are executed is called a *scheduling algorithm* or *scheduler*. The problem in our context is to define a scheduling algorithm to select the next process among an object's method activations when the object is idle. For this purpose, we shall represent (runtime) *processes* as a datatype in Real-Time ABS in such a way that we can express *scheduling algorithms* as functions inside the language.

***Processes.*** Real-time systems are characterized by computational activities with timing constraints that must be met in order to achieve the desired behavior. A typical timing constraint on a process is the *deadline*, which represents the time before which the process should complete its execution. Depending on the consequence of a missed deadline, a process is usually called *hard* if a completion after its deadline can cause catastrophic consequences on the system and *soft* if missing its deadline decreases the performance of the system but does not jeopardize the system's correct behavior. In general, real-time processes are characterized by a number of well-known parameters. In the context of Real-Time ABS models, these parameters are either part of the *specification* supplied by the modeler, or else *measured* at runtime:

- *Arrival Time* $r$ (measured at runtime): the time when a process becomes ready for execution.

- *Computation Time* $c$ (specified together with the method definition): the time needed by the object to execute the process without interruption,
- *Relative Deadline* $d$ (specified at method call site): the time before which a process should be completed to avoid damage or performance degradation. It is specified with respect to the arrival time.
- *Start Time* $s$ (measured at runtime): the time at which a process starts its execution.
- *Finish Time* $f$ (measured at runtime): the time at which a process finishes its execution.
- *Criticality* $crit$ (specified at method call site): a parameter related to the consequence of missing a deadline (hard or soft).
- *Value* $v$ (specified / set during process execution): a value representing the relative importance of a process with respect to other processes.

The following further parameters can be calculated from the parameters above:

- *Absolute Deadline* ($D$): similar to the relative deadline, but with respect to time *zero* (i.e., $D = r + d$).
- *Response Time* ($R$): the difference between the finish time and the arrival time (i.e., $R = f - r$).
- *Lateness* ($L$): the possible delay of the process completion with respect to the absolute deadline (i.e., $L = f - D$). Note that if the completion happens before the deadline then $L$ is negative.
- *Tardiness* ($E$) or exceeding time: the time a process stays active after its deadline (i.e., $E = max(0, L)$).
- *Laxity* ($X$): the maximum time a process can delay its activation to complete within its deadline (i.e., $X = r - c$).

These parameters can be used to define different scheduling policies, scheduling algorithms, as well as for performance evaluation; e.g., the *lateness* can be used to observe the optimality of a given scheduling policy or algorithm.

***Processes in Real-Time ABS.*** Based on the above parameters, we define datatypes for *time* `Time`, *process identifiers* `Pid`, and *processes* `Process` as follows:

```
data Time = Time(Rat);
data Pid ;
data Process = Proc(Pid pid, String m, Time r, Duration c,
                    Duration d, Time s, Time f, Bool crit, Int v);
```

Here, `m` represents the method name associated with the process, `r` represents the arrival time recorded when the method is bound, `c` represents the computation time or cost (the cost heuristics is an optional annotation to method declarations), `d` the relative deadline (an optional annotation of the method call), `s` the starting time, `f` the finishing time, `crit` the criticality (an optional annotation of the method call) and `v` the value representing the relative importance of the process (this is an assignable local variable in the process). `pid` is a token identifying the underlying process and does not have a syntax defined at the language level; this ensures that user-defined scheduling algorithms can under no circumstance choose a non-existing process to be run. For two `Time` values $t_1$ and $t_2$, denote by $t_1 \leq t_2$ their comparison defined by comparing their rational arguments, and by $t_1 - t_2$ their subtraction which is of type Duration. For easier readability, we also define observer functions for the process parameters, e.g. for the process name:

```
def String name(Process p) =
    case p {
      Proc(_,m,_,_,_,_,_,_,_) => m;
    };
```

The observer functions for process id, arrival time, cost, deadline, value, etc. are defined in the same way and are used in defining the scheduling policies. The lifting of runtime method activations into the datatype `Process` is explained in Section 5.4.1.

### 5.3.1 General Scheduling Policies

During the modeling of software systems, it is important to consider which scheduling policy will provide the best application-level performance for the corresponding system. There is no universal "best" scheduling policy, and it is possible to both define new scheduling policies and to combine different scheduling policies. In Real-Time ABS, a scheduler algorithm is a function which, given a non-empty list of processes, returns the next process to be executed. Some examples of schedulers in Real-Time ABS are given below.

**Example 2.** *The* default *scheduler, which selects the first process in the list for execution, is defined as follows:*

```
def Process default (List<Process> l) = head(l);
```

**Example 3.** *Many well-known scheduling policies fit into a general pattern: they define a total ordering of processes depending on some ordering predicate. Such scheduling policies can be realized in Real-Time ABS using a pattern consisting of a function* `scheduler` *which takes a non-empty list of processes and returns the best process according to a comparison function comp implementing the ordering predicate. A helper function* `schedulerH` *compares the best process found so far to the remaining list of processes.*

```
def Process scheduler (List<Process> l) = schedulerH(head(l), tail(l));

def Process schedulerH(Process p1, List<Process> l1) =
    case l1 {
      Nil => p1;
      Cons(p2,l2) =>
        if (comp(p1, p2))
        then schedulerH(p1,l2)
        else schedulerH(p2,l2)
    };
```

*In order to realize a specific scheduling policy, we follow the pattern above and replace comp*`(p1,p2)` *with a suitable comparison function for the desired policy (since Real-Time ABS does not support first-class function arguments, we open-code the concrete algorithm).*

- Earliest deadline first (edf) *is a dynamic scheduling policy that selects processes according to their absolute deadline. Processes with earlier deadlines will be executed with a higher priority.*

  *The scheduling function* `edf(l)` *is obtained by using the following comparison function:*

  ```
  def Bool comp_edf(Process p1, Process p2) = lte(deadline(p1), deadline(p2));
  ```

- First in, first out (fifo) *selects processes according to their arrival order. The scheduling function* `fifo(l)` *uses the following comparison function:*

  ```
  def Bool comp_fifo(Process p1, Process p2) = arrival(p1) ≤ arrival(p2);
  ```

- Fixed priority (fp) *selects the process with the greatest fixed assigned priority from the process queue. In our setting, it is natural to fix the priority of processes depending on the name of the activated methods. Let* `fp(l)` *be the scheduler defined as above, but using the following comparison function:*

  ```
  def Bool comp_fp(Process p1, Process p2) = weight(name(p1)) ≥ weight(name(p2));

  def Int weight(String s) =
      case s {
        "method1" => v1;
        "method2" => v2; ...
      };
  ```

- Dynamic priority (dp) *is similar to* fp *but the priority is not fixed. Rather, the priority depends on the $v$ (value) attribute of the processes, which can be modified by the process itself during execution.* `dp(l)` *is the scheduler defined as above using the following comparison function (where lower value gives higher priority):*

```
def Bool comp_dp(Process p1, Process p2) = value(p1) ≤ value(p2);
```

- Shortest job first (sjf) *selects the process with the least remaining computation time (cost) from the queue. Let* `sjf(l)` *be the scheduler defined as above, which uses the following comparison function:*

```
def Bool comp_sjf(Process p1, Process p2) = lte(cost(p1), cost(p2));
```

Different scheduling policies may be combined. This may be illustrated by the following example.

**Example 4** (Combined sjf and dp)**.** *We consider a combination of* sjf *and* dp*, where the process with the lowest cost is selected among the processes with the highest priority. In this case, we define a filter* `highPri` *and call* `sjf` *on the filtered list of processes:*

```
def List<Process> highPri(List<Process> l1, List<Process> l2) =
    case l2 {
      Nil => l1;
      Cons(h,t) =>
         if (l1 == Nil)
         then highPri(Cons(h,Nil),t)
         else if (comp_dp (head(l1),h))
             then if (value(head(l1)) == value(h))
                 then highPri(Cons(h,l1), t)
                 else highPri(l1,t)
             else highPri(Cons(h,Nil),t)
    };

def Process sjfdp(List<Process> l) = sjf(highPri(Nil,l));
```

### 5.3.2 Conditional Scheduler

Here we show how to define a scheduler which changes the priority of a set of processes depending on the length of the process queue.

**Example 5.** *We consider an object which has a set of processes to be scheduled by some scheduling algorithm* `scheduler`*. Some of these processes, e.g., a load balancer, need to have high priority when there is congestion inside the object. Let $l$ be a non-empty process queue, $n$ an integer representing the queue length limit (i.e., the process priority changes when the size of the queue $l$ grows beyond $n$), and let ccp denote*

*the set of method names for the processes with conditionally changing priority. Let* `filter` *be a function which filters processes with names contained in ccp from the process queue l.*

```
def Process condScheduler(List<Process> l, List<String> ccp, Int n) =
    if (length(l) ≤ n || filter(ccp,l) = Nil)
    then scheduler(l)
    else scheduler(filter(ccp,l))
```

Note that the list *ccp* of method names as well as the the queue length limit $n$ are highly application dependent. In fact, these parameters of the scheduling function can be state dependent in Real-Time ABS.

### 5.3.3 Scheduling Annotations in Real-Time ABS

In Real-Time ABS, there are three levels at which the scheduling policy of a concurrent object can be determined. There is a default system-level scheduling policy, as defined in Example 2. This default scheduling policy may be overridden for all instances of a class by providing a scheduling annotation for the class definition. This class-level scheduling policy may in turn be overridden for a specific instance by a scheduling annotation at the object creation statement. In this way, different instances of a class may have different scheduling policies, for example to improve the performance for different application specific user scenarios. In a scheduling annotation, the keyword **`queue`** is used to refer to the ABS datatype representation of the runtime process queue of the scheduled object. Additional formal parameters of a scheduling function are bound to object attributes. This means that the object state can influence the scheduling, Example 9 in Section 5.5 gives an example.

### 5.3.4 Monitors with Signal and Continue Discipline

A monitor [4, 21] is a high-level synchronization mechanism which protects shared state by only allowing access to the shared state through a given set of methods. The defining characteristic of a monitor is that its methods are executed with mutual exclusion; similar to processes in a Real-Time ABS object, at most one process may be executing at any time inside the monitor. Monitors use *condition variables* to delay processes until the monitors's state satisfies some Boolean condition. This condition variable is also used to awaken a delayed process when the condition becomes true. The value of a *condition variable* is associated with a *fifo* queue of delayed processes. Processes which are blocked on *condition variables* get awakened by a signal call.

In Real-Time ABS, objects may be regarded as abstract monitors without the need for explicit signaling, which is guaranteed by the semantics and need not be the

responsibility of the modeler. However, the order in which processes are activated in the process queue of a Real-Time ABS object does not guarantee a *fifo* ordering of the queue. When explicit signaling is desired, monitors with different signaling disciplines can be encoded. We now show how such an ordering may be explicitly enforced independent of the scheduling policy of the object (following [25]), and then how this synchronization code can be simplified by specifying a *fifo* scheduling policy for the object.

**Example 6.** *Consider a class implementing a general monitor with the* signal and continue *discipline [4]; for simplicity the example is restricted to one condition variable. Without any assumption about the scheduling strategy for the process queue of the monitor, we need to introduce synchronization code to ensure that suspended processes are activated following a* fifo *ordering. This can be achieved using a triple* $\langle s, d, q \rangle$ *of natural numbers; where* $s$ *represents available signals to the condition variable,* $d$ *the number of the delayed process in the queue of the condition variable, and* $q$ *the number of delayed processes that have been reactivated.*

```
interface Monitor {
   Unit wait();
   Unit signal();
   Unit signalAll();
}

class MonitorImp() implements Monitor {
    Int s = 1;
    Int d = 0;
    Int q = 0;

    Unit wait() {
       Int myturn = d+1;
       d = d+1;
       await (s>0 ∧ q+1==myturn);
       s=s−1;
       q=q+1;
    }

    Unit signal() {
       if (d>q) {
          s=s+1;
       }
    }

    Unit signalAll() {
       s=d−q;
    }
}
```

**Example 7.** *By ensuring a* fifo *scheduling policy for the monitor object, the synchronization code of Example 6 can be significantly simplified. Let* $l$ *represent the length of the queue of waiting processes and* $s$ *the number of available signals as before.*

```
[Scheduler: fifo(queue)]
class SimpleMonitorImp() implements Monitor {
    Int s = 1;
    Int l = 0;

    Unit wait() {
        l = l+1;
        await s>0;
        s=s−1;
        l=l−1;
    }

    Unit signal() {
        if (l>0) {
            s=s+1;
        }
    }

    Unit signalAll() {
        s = l;
    }
}
```

This is an example of how a specific scheduling policy can influence object execution semantics.

## 5.4 Semantics

This section presents the operational semantics of Real-Time ABS as a transition system in an SOS style [34]. Rules apply to subsets of configurations (the standard context rules are not listed). For simplicity we assume that configurations can be reordered to match the left hand side of the rules (i.e., matching is modulo associativity and commutativity as in rewriting logic [29]). A run is a possibly nonterminating sequence of rule applications. When auxiliary functions are used in the semantics, these are evaluated in between the application of transition rules in a run.

### 5.4.1 Runtime Configurations

The runtime syntax is given in Fig. 5.3. We extend expressions $e$ with the runtime syntax **case2** $v$ $\{\overline{br}\}$, statements with **duration2**$(d, d)$, and values $v$ with identifiers for objects and futures. For simplicity, we assume that all class and method declarations, as well as assignments for object creation and method calls are annotated as explained in Section 5.2 (i.e., the compiler inserts defaults and orders annotations where appropriate).

$$
\begin{array}{rcl}
e & ::= & \texttt{case2}\ v\ \{\overline{br}\} \mid \ldots \\
v & ::= & o \mid f \mid \ldots \\
s & ::= & \texttt{duration2}(d_1, d_2) \mid \ldots \\
cn & ::= & \epsilon \mid obj \mid msg \mid fut \mid cn\ cn \\
tcn & ::= & cn\ \mathrm{clock}(t) \\
fut & ::= & f \mid \mathrm{fut}(f, v) \\
\sigma & ::= & x \mapsto v \mid \sigma \circ \sigma \\
obj & ::= & \mathrm{ob}(o, e, \sigma, pr, q) \\
pr & ::= & \{\sigma | s\} \mid \mathrm{idle} \\
msg & ::= & m(o, \overline{v}, f, d, c, t) \\
q & ::= & \epsilon \mid pr \mid q \circ q
\end{array}
$$

Figure 5.3: Runtime syntax; here, $o$ and $f$ are object and future identifiers, $d$ and $c$ are the deadline and cost annotations.

*Configurations $cn$* are sets of objects, invocation messages, and futures. A *timed configuration $tcn$* adds a global clock $clock(t)$ to a configuration (where $t$ is a value of type Time). The global clock is used to record arrival and finishing times for processes. Timed configurations live inside curly brackets; thus, in $\{cn\}$, $cn$ captures the *entire* runtime configuration of the system. The associative and commutative union operator on (timed) configurations is denoted by whitespace and the empty configuration by $\varepsilon$.

An *object obj* is a term $ob(o, e, \sigma, pr, q)$ where $o$ is the object's identifier, $e$ is an expression of type Process representing a *scheduling policy*, $\sigma$ a substitution representing the object's fields, $pr$ is an (active) process, and $q$ a *pool of processes*. A *substitution $\sigma$* is a mapping from variable names $x$ to values $v$. For substitutions and process pools, concatenation is denoted by $\sigma_1 \circ \sigma_2$ and $q_1 \circ q_2$, respectively.

In an *invocation message* $m(o, \overline{v}, f, d, c, t)$, $m$ is the method name, $o$ the callee, $\overline{v}$ the call's actual parameter values, $f$ the future to which the call's result is returned, $d$ and $c$ are the provided deadline and cost of the call, and $t$ is a time stamp recording the time of the call. A *future* is either an identifier $f$ or a term $fut(f, v)$ with an identifier $f$ and a reply value $v$. For simplicity, classes are not represented explicitly in the semantics, but may be seen as static tables of object layout and method definitions.

***Processes and Process Lifting.*** A *process* $\{\sigma|s\}$ consists of a substitution $\sigma$ of local variable bindings and a list $s$ of statements, or it is *idle*. By default, the local variables of a process include the variables `method` of type String, `arrival` of type Time, `cost` of type Duration, `deadline` of type Duration, `start` of type Time, `finish` of type Time, `critical` of type Bool, `value` of type Int, and `destiny` of type Name. Consequently, we can define a function *lift* which transforms the runtime representation of a process into the Real-Time ABS datatype of processes and a function *select* which returns the process corresponding to a given process identifier

$$
\begin{aligned}
&[\![b]\!]_\sigma = b\\
&[\![x]\!]_\sigma = \sigma(x)\\
&[\![v]\!]_\sigma = v\\
&[\![Co(\overline{e})]\!]_\sigma = Co([\![\overline{e}]\!]_\sigma)\\
&[\![\mathtt{deadline}]\!]_\sigma = \sigma(\mathtt{deadline})\\
&[\![fn(\overline{e})]\!]_\sigma = \begin{cases} [\![e_{fn}]\!]_{\overline{x}\mapsto\overline{v}} & \text{if } \overline{e} = \overline{v}\\ [\![fn([\![\overline{e}]\!]_\sigma)]\!]_\sigma & \text{otherwise}\end{cases}\\
\\
&[\![\mathtt{case}\ e\ \{\overline{br}\}]\!]_\sigma = [\![\mathtt{case2}\ [\![e]\!]_\sigma\ \{\overline{br}\}]\!]_\sigma\\
&[\![\mathtt{case2}\ t\ \{p \Rightarrow e; \overline{br}\}]\!]_\sigma = \begin{cases} [\![e]\!]_{\sigma\circ match(p,t)} & \text{if } match(p,t) \neq \bot\\ [\![\mathtt{case2}\ t\ \{\overline{br}\}]\!]_\sigma & \text{otherwise}\end{cases}
\end{aligned}
$$

Figure 5.4: The evaluation of functional expressions.

in a process queue, as follows:

$$
\begin{aligned}
lift(\{\sigma|s\}) &= \mathtt{Proc}(\sigma(\mathtt{destiny}), \sigma(\mathtt{method}), \sigma(\mathtt{arrival}), \sigma(\mathtt{cost}), \sigma(\mathtt{deadline}),\\
&\qquad \sigma(\mathtt{start}), \sigma(\mathtt{finish}), \sigma(\mathtt{crit}), \sigma(\mathtt{value}))\\
select(pid, \varepsilon) &= \text{idle}\\
select(pid, \{\sigma|s\} \circ q) &= \begin{cases} \{\sigma|s\} & \text{if } \sigma(\mathtt{destiny}) = pid\\ select(pid, q) & \text{otherwise}\end{cases}
\end{aligned}
$$

The value of `destiny` is guaranteed to be unique, and is used to identify processes at the Real-Time ABS level.

### 5.4.2 A Reduction System for Expressions

The strict evaluation $[\![e]\!]_\sigma$ of functional expressions $e$, given in Fig. 5.4, is defined inductively over the data types of the functional language and is mostly standard, hence this subsection only contains brief remarks about some of the expressions. Let $\sigma$ be a substitution which binds the name `deadline` to a duration value. For every (user-defined) function definition

$$\mathtt{def}\ T\ fn(\overline{T\ x}) = e_{fn},$$

the evaluation of a function call $[\![fn(\overline{e})]\!]_\sigma$ reduces to the evaluation of the corresponding expression $[\![e_{fn}]\!]_{\overline{x}\mapsto\overline{v}}$ when the arguments $\overline{e}$ have already been reduced to ground terms $\overline{v}$. (Note the change in scope. Since functions are defined independently of the context where they are used, we here assume that the expression $e$ does not contain free variables and the substitution $\sigma$ does not apply in the evaluation of $e$.) In the case of pattern matching, variables in the pattern $p$ may be bound to argument values in $v$. Thus the substitution context for evaluating the right hand side $e$ of the branch $p \Rightarrow e$ extends the current substitution $\sigma$ with bindings that occurred during the

pattern matching. Let the function $match(p, v)$ return a substitution $\sigma$ such that $\sigma(p) = v$ (if there is no match, $match(p, v) = \bot$).

### 5.4.3 A Transition System for Timed Configurations

*Evaluating Guards.* Given a substitution $\sigma$ and a configuration $cn$, we lift the evaluation function for functional expressions and denote by $[\![g]\!]_\sigma^{cn}$ a evaluation function which reduces guards $g$ to data values (the state configuration is needed to evaluate future variables). Let $[\![g_1 \wedge g_2]\!]_\sigma^{cn} = [\![g_1]\!]_\sigma^{cn} \wedge [\![g_2]\!]_\sigma^{cn}$, $[\![\mathbf{duration}(b, w)]\!]_\sigma^{cn} = [\![b]\!]_\sigma \leq 0$, $[\![x?]\!]_\sigma^{cn} = \text{true}$ if $[\![x]\!]_\sigma = f$ and $fut(f, v) \in cn$ for some value $v$ (otherwise $f \in cn$ and we let $[\![x?]\!]_\sigma^{cn} = \text{false}$), and $[\![b]\!]_\sigma^{cn} = [\![b]\!]_\sigma$.

*Auxiliary functions.* If $T$ is the return type of a method $m$ in a class $C$, we let $bind(m, o, \overline{v}, f, d, b, t)$ return a process resulting from the activation of $m$ in the class of $o$ with actual parameters $\overline{v}$, callee $o$, associated future $f$, deadline $d$, and criticality $b$ at time $t$. If binding succeeds, this process has a local variable `destiny` of type `fut`$\langle T \rangle$ bound to $f$, the method's formal parameters are bound to $\overline{v}$, and the reserved variables `deadline` and `critical` are bound to $d$ and $b$, respectively. Furthermore, `arrival` is bound to $t$ and `cost` to $[\![e]\!]_{\overline{x} \mapsto \overline{v}}$ (or to the default 0 if no annotation is provided for the method). The function $atts(C, \overline{v}, o)$ returns the initial state of an instance of class $C$, in which the formal parameters are bound to $\overline{v}$ and the reserved variables `this` is bound to the object identity $o$. The function $init(C)$ returns an activation of the *init* method of $C$, if defined. Otherwise it returns the *idle* process. The predicate $fresh(n)$ asserts that a name $n$ is globally unique (where $n$ may be an identifier for an object or a future).

*Transition rules* transform state configurations into new configurations, and are given in Fig. 5.5 and Fig. 5.6. We denote by $a$ the substitution which represents the attributes of an object and by $l$ the substitution which represents the local variable bindings of a process. In the semantics, different assignment rules are defined for side effect free expressions (Assign1 and Assign2), object creation (New-Object), method calls (Async-Call), and future dereferencing (Read-Fut). Rule Skip consumes a **`skip`** in the active process. Here and in the sequel, the variable $s$ will match any (possibly empty) statement list. We denote by *idle* a process with an empty statement list. Rules Assign1 and Assign2 assign the value of expression $e$ to a variable $x$ in the local variables $l$ or in the fields $a$, respectively. Rules Cond1 and Cond2 cover the two cases of conditional statements in the same way. (We omit the rule for **`while`**-loops which unfolds into the conditional.)

*Scheduling.* Two operations manipulate a process pool $q$; $pr \circ q$ adds a process $pr$ to $q$ and $q \setminus pr$ removes $pr$ from $q$. If $q$ is a pool of processes, $\sigma$ a substitution, $t$ a time value, and $cn$ a configuration, we denote by $ready(q, \sigma, cn)$ the subset of processes from $q$ which are ready to execute (in the sense that the processes will not directly

(Skip)
$$\mathrm{ob}(o,p,a,\{l \mid \texttt{skip}; s\}, q) \to \mathrm{ob}(o,p,a,\{l \mid s\}, q)$$

(Suspend)
$$\mathrm{ob}(o,p,a,\{l \mid \texttt{suspend}; s\}, q) \to \mathrm{ob}(o,p,a,\mathrm{idle},\{l \mid s\} \circ q)$$

(Tick)
$$\frac{0 < d \leq mte(cn)}{\{cn\} \to \{adv(cn,d)\}}$$

(Activation)
$$\frac{q' = bind(m,o,\bar{v},f,d,b,t) \circ q}{\mathrm{ob}(o,p,a,pr,q)\ m(o,\bar{v},f,d,b,t) \to \mathrm{ob}(o,p,a,pr,q')}$$

(Read-Fut)
$$\frac{f = [\![e]\!]_{a \circ l}}{\mathrm{ob}(o,p,a,\{l \mid x = e.\texttt{get}; s\}, q)\ \mathrm{fut}(f,v) \to \mathrm{ob}(o,p,a,\{l \mid x = v; s\}, q)\ \mathrm{fut}(f,v)}$$

(Await1)
$$\frac{[\![e]\!]^{cn}_{a \circ l}}{\{\mathrm{ob}(o,p,a,\{l \mid \texttt{await}\ e; s\}, q)\ cn\} \to \{\mathrm{ob}(o,p,a,\{l \mid s\}, q)\ cn\}}$$

(Assign1)
$$\frac{x \in dom(l)}{\mathrm{ob}(o,p,a,\{l \mid x = e; s\}, q) \to \mathrm{ob}(o,p,a,\{l[x \mapsto [\![e]\!]_{a \circ l}] \mid s\}, q)}$$

(Await2)
$$\frac{\neg[\![e]\!]^{cn}_{a \circ l}}{\{\mathrm{ob}(o,p,a,\{l \mid \texttt{await}\ e; s\}, q)\ cn\} \to \{\mathrm{ob}(o,p,a,\{l \mid \texttt{suspend}; \texttt{await}\ e; s\}, q)\ cn\}}$$

(Assign2)
$$\frac{x \notin dom(l)}{\mathrm{ob}(o,p,a,\{l \mid x = e; s\}, q) \to \mathrm{ob}(o,p,a[x \mapsto [\![e]\!]_{a \circ l}],\{l \mid s\}, q)}$$

Figure 5.5: The semantics of Real-Time ABS (1).

suspend or block the object [25]).

Scheduling is captured by the rule Schedule, which applies when the active process is *idle* and schedules a new process for execution if there are ready processes in the process pool $q$. We utilize a scheduling policy as explained in Section 5.3: in an object $ob(o, p, \sigma, idle, q)$, $p$ is an expression representing the user-defined scheduling policy. This policy selects the process to be scheduled among the ready processes of the pool $q$.

In order to apply the scheduling policy $p$, which is defined for the datatype Process in Real-Time ABS, to the runtime representation $q$ of the process pool, we lift the processes in $q$ to values of type Process. Let the function *liftall* recursively transform a pool $q$ of processes to a value of type List⟨Process⟩ by repeatedly applying *lift* to the processes in $q$. The process identifier of the scheduled process is used to *select* the runtime representation of this process from $q$.

Note that in order to evaluate guards on futures, the configuration $cn$ is passed to the *ready* function. This explains the use of brackets in the rules, which ensures that $cn$ is bound to the rest of the global system configuration. The same approach is used to evaluate guards in the rules Await1 and Await2 below.

(Schedule)

$$\frac{q' = \mathit{ready}(q, a, cn) \qquad pr = \mathit{select}(pid, q) \qquad q' \neq \emptyset \quad pid = [\![\mathit{procid}(p)]\!]_{a[\mathtt{queue} \mapsto \mathit{liftall}(q')]}}{\{\mathrm{ob}(o, p, a, \mathrm{idle}, q)\ cn\} \to \{\mathrm{ob}(o, p, a, pr, (q \setminus pr))\ cn\}}$$

(New-Object)

$$\frac{an = \mathtt{Scheduler:}\ p' \quad \mathit{fresh}(o') \qquad pr = \mathit{init}(C) \quad a' = \mathit{atts}(C, [\![\bar{e}]\!]_{a \circ l}, o', c)}{\mathrm{ob}(o, p, a, \{l|[an]\ x = \mathbf{new}\ C(\bar{e}); s\}, q) \to \mathrm{ob}(o, p, a, \{l|x = o'; s\}, q)\ \mathrm{ob}(o', p', a', pr, \emptyset)}$$

(Duration1)

$$\frac{d_1 = [\![e_1]\!]_{a \circ l} \quad d_2 = [\![e_2]\!]_{a \circ l}}{\mathrm{ob}(o, p, a, \{l \mid \mathbf{duration}(e_1, e_2); s\}, q) \to \mathrm{ob}(o, p, a, \{l \mid \mathbf{duration2}(d_1, d_2); s\}, q)}$$

(Duration2)

$$\frac{d_1 \leq 0}{\mathrm{ob}(o, p, a, \{l \mid \mathbf{duration2}(d_1, d_2); s\}, q) \to \mathrm{ob}(o, p, a, \{l \mid s\}, q)}$$

(Async-Call)

$$\frac{\mathit{fresh}(f) \quad an = \mathtt{Deadline:}\ d,\ \mathtt{Critical:}\ b}{\mathrm{ob}(o, p, a, \{l \mid [an]\ x := e!m(\bar{e}); s\}, q)\ \mathrm{clock}(t) \to \mathrm{ob}(o, p, a, \{l \mid x := f); s\}, q)\ \mathrm{clock}(t)\ m([\![e]\!]_{a \circ l}, [\![\bar{e}]\!]_{a \circ l}, f, d, b, t)\ f}$$

(Return)

$$\frac{f = l(\mathtt{destiny})}{\mathrm{ob}(o, p, a, \{l \mid \mathbf{return}(e); s\}, q)\ \mathrm{clock}(t)\ f \to \mathrm{ob}(o, p, a, \{l \mid \mathtt{finish} = t\}, q)\ \mathrm{clock}(t)\ \mathrm{fut}(f, [\![e]\!]_{a \circ l})}$$

(Cond2)

$$\frac{\neg [\![e]\!]_{a \circ l}}{\mathrm{ob}(o, p, a, \{l|\mathbf{if}\ e\ \{s_1\}\ \mathbf{else}\ \{s_2\}; s\}, q) \to \mathrm{ob}(o, p, a, \{l|s_2; s\}, q)}$$

(Cond1)

$$\frac{[\![e]\!]_{a \circ l}}{\mathrm{ob}(o, p, a, \{l|\mathbf{if}\ e\ \{s_1\}\ \mathbf{else}\ \{s_2\}; s\}, q) \to \mathrm{ob}(o, p, a, \{l|s_1; s\}, q)}$$

Figure 5.6: The semantics of Real-Time ABS (2).

Rule Suspend suspends the active process to the process pool, leaving the active process *idle*. Rule Await1 consumes the **await** $g$ statement if $g$ evaluates to true in the current state of the object, rule Await2 adds a suspend statement in order to suspend the process if the guard evaluates to false.

In rule Activation the function $\mathrm{bind}(m, o, \bar{v}, f, d, c, b, t)$ binds a method call to object $o$ in the class of $o$. This results in a new process $\{l|s\}$ which is placed in the queue, where $l(\mathtt{destiny}) = f$, $l(\mathtt{method}) = m$, $l(\mathtt{arrival}) = t$, $l(\mathtt{cost}) = c$, $l(\mathtt{deadline}) = d$, $l(\mathtt{start}) = 0$, $l(\mathtt{finish}) = 0$, $l(\mathtt{crit}) = b$, $l(\mathtt{value}) = 0$, and where the formal parameters of $m$ are bound to $\bar{v}$.

*Durations.* A statement $\mathbf{duration}(e_1, e_2)$ is reduced to the runtime statement $\mathbf{duration2}(d_1, d_2)$, in which the expressions $e_1$ and $e_2$ have been reduced to duration values. This statement blocks execution on the object until the best case execution time has passed; i.e., until at least the duration $d_1$ has passed. Remark that time

$$
\begin{aligned}
&mte(cn_1\ cn_2) = min(mte(cn_1), mte(cn_2))\\
&mte(\mathrm{ob}(o,p,a,pr,q)) = \begin{cases} mte(pr) & \text{if } pr \neq \text{idle}\\ mte(q) & \text{if } pr = \text{idle}\end{cases}\\
&mte(q_1,q_2) = min(mte(q_1), mte(q_2))\\
&mte(\{l|s\}) = \begin{cases} w & \text{if } s = \texttt{duration2}(b,w); s_2\\ mte(g) & \text{if } s = \texttt{await}\ g; s_2\\ 0 & \text{if } s \text{ is enabled}\\ \infty & \text{otherwise}\end{cases}\\
&mte(g) = \begin{cases} max(mte(g_1), mte(g_2)) & \text{if } g = g_1 \wedge g_2\\ w & \text{if } g = \texttt{duration}(b,w)\\ 0 & \text{if } g \text{ evaluates to true}\\ \infty & \text{otherwise}\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&adv(cn_1\ cn_2, d) = adv(c_1, d)\ adv(c_2, d)\\
&adv(\mathrm{ob}(o,p,a,pr,q), d) = \mathrm{ob}(o,p,a,adv(pr,d), adv(q,d))\\
&adv((q_1,q_2),d) = adv(q_1,d), adv(q_2,d)\\
&adv(\{l|s\},d) = \{l[\texttt{deadline} \mapsto l(\texttt{deadline}) - d]|adv(s,d)\}\\
&adv(s,d) = \begin{cases} \texttt{duration2}(b-d, w-d) & \text{if } s = \texttt{duration2}(b,w)\\ \texttt{await}\ adv(g,d) & \text{if } s = \texttt{await}\ g\\ adv(s_1,d) & \text{if } s = s_1; s_2\\ s & \text{otherwise}\end{cases}\\
&adv(g,d) = \begin{cases} adv(g_1,d) \wedge adv(g_2,d) & \text{if } g = g_1 \wedge g_2\\ \texttt{duration}(b-d, w-d) & \text{if } g = \texttt{duration}(b,w)\\ g & \text{otherwise}\end{cases}
\end{aligned}
$$

Figure 5.7: Functions controlling the advancement of time. Trivial cases for terms *msg*, *fut* have been omitted.

cannot pass beyond duration $d_2$ before the statement has been executed (see below).

*Method Calls.* Rule Async-Call sends an invocation message to $[\![e]\!]_{a\circ l}$ with the unique identity $f$ of a new future (since fresh$(f)$), the method name $m$, and parameter values $\overline{v}$. The identifier of the new future is placed in the configuration, and is bound to a return value in Return. The annotations are used to provide a deadline and a criticality which are passed to the callee with the invocation message. (The global clock provides a time stamp for the call.) Rule Return places the evaluated return expression in the future associated with the *destiny* variable of the process, and ends execution after recording the time of process completion in the `finish` variable. Rule Read-Fut dereferences the future fut$(f,v)$. Note that if the future lacks a return value, the reduction in this object is *blocked*.

*Object creation.* Rule New-Object creates a new object with a unique identifier $o'$. The object's fields are given default values by $atts(C, [\![\overline{e}]\!]_{a\circ l}, o', c)$, extended

```
interface Server {
 Bool request(String job, Rat bc, Rat wc);
}

data Log = Log(String job, Time completiontime, Duration jobdeadline);

[Scheduler: sjf(queue)]
class ServerImp implements Server {
    List<Log> history = Nil;

    [Cost: Duration(wc)]
    Bool request(String job, Rat bc, Rat wc) {
        duration(bc,wc);
        history = Cons(Log(job, now, deadline),history);
        return (durationValue(deadline) > 0);
    }
 }

interface Client { }

class ClientImp (String job, Int cycles, Int frequency, Duration bc, Duration wc,
                 Duration limit, Server s) implements Client {
    Int replies = 0;
    Int successes = 0;

    Unit run() { await duration(frequency,frequency);
        [Deadline: limit]
        Fut<Bool> res = s!request(job, durationValue(bc),durationValue(wc));
        cycles = cycles - 1;
        if (cycles>0){
             this!run();
        }
        await res?;
        replies = replies + 1;
        Bool result = res.get;
        if (result){
             successes = successes+1;
       }
    }
}

{
   // Main block:
   Server s = new ServerImp();
   Client photo = new ClientImp("Photo",10,15, Duration(2),Duration(2),Duration(40),s);
   Client video = new ClientImp("Video",4,40, Duration(15),Duration(15),Duration(80),s);
   ...
}
```

Figure 5.8: A model of photo and video processing. The server class as shown uses *sjf* (shortest job first) scheduling.

with the actual values $\overline{e}$ for the class parameters (evaluated in the context of the creating process) and $o'$ for **`this`**. In order to instantiate the remaining attributes, the process *pr* is active (we assume that this process reduces to *idle* if *init*($C$) is unspecified in the class definition, and that it asynchronously calls `run` if the latter is specified). The object gets the scheduler in the annotation *an* (which is copied from the class or system default if a scheduler annotation is not provided).

*Time advance.* Rule Tick specifies how time can advance in the system. We adapt the approach of Real-Time Maude [31,32] to Real-Time ABS and specify a global time which advances uniformly throughout the global configuration *cn*, combined with two auxiliary functions: $adv(cn, d)$ specifies how the advance of time with a duration $d$ affects different parts of the configuration *cn*, and $mte(cn)$ defines the maximum amount that global time can advance. At any time, the system can advance by a duration $d \leq mte(cn)$. However, we are not interested in advancing time by a duration 0, which would leave the system in the same state.

The auxiliary functions *adv* and *mte* are defined in Fig. 5.7. Both have the whole configuration as input but consider mainly objects since these exhibit time-dependent behavior. The function *mte* calculates the maximum time increment such that no "interesting" occurrence (i.e., worst-case duration expires, duration guard passes) will be missed in any object. Observe that for statements which are not time-dependent, the maximum time elapse is 0 if the statement is enabled, since these statements are instantaneous, and infinite if not enabled, since time may pass when the object is blocked. Hence, *mte* returns the minimum time increment that lets an object become "unstuck", either by letting its active process continue or enabling one of its suspended processes. The function *adv* updates the active and suspended processes of all objects, decrementing all `deadline` values as well as the values in **`duration`** statements and **`duration`** guards at the head of the statement list in processes.

## 5.5 Case Studies and Simulation Results

A tool for Real-Time ABS is implemented, extending the existing ABS interpreter and tool chain. The parser, type-checker, and code generator have been implemented in Java using the JastAdd toolkit. Real-Time ABS models run on top of the Maude rewrite engine [14], with a model-independent interpreter almost directly implementing the semantics of Sec. 5.4. Code editing is supported in either the Emacs editor or the Eclipse integrated development environment, both of which support the ABS language after installing a plugin.

The examples below are based on the common scenario of a *server* receiving a series of *method calls* from an environment over time. These method calls result in method activations which have associated deadlines and costs. The examples further demonstrate the use of the Real-Time ABS tool.

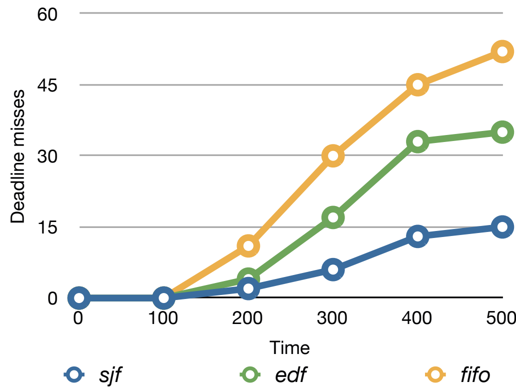


Figure 5.9: Simulation results for Example 8: Comparison of the number of missed deadlines with respect to time for *fifo*, *edf*, and *sjf* schedulers.

**Example 8.** *Consider the ABS model of a service, which is used by clients by calling the* `request` *method of a* `Server` *object (Fig. 5.8). For simplicity, we abstract from the specific functionality of our service (e.g., resizing photos or video of varying sizes) and let the* `request` *method of a server have a certain* `duration` *instead which is given as a parameter to the method. This duration reflects the cost of execution of the service in terms of its inputs best case* `bc` *and worst case* `wc` *execution time. A request to the server is successful (captured by the return value of the method) if it can be handled within the deadline which is given as an annotation at the call site in the* `ClientImp` *implementation.*

*The class* `ClientImp` *takes a number of jobs and dispatches them with a certain frequency to the server's* `request` *method with the given deadline. The class* `ServerImp` *contains a field* `history` *that is recording the scheduling sequence of the jobs and their lateness. The number of received and successful responses to* `request` *calls are recorded in the two variables* `replies` *and* `successes` *in the* `ClientImp` *class.*

*We simulate the model with a usage scenario in which objects for* photo clients *and* video clients *send a total of 70 jobs to the server. Approximately 70% of the jobs are cheap (i.e., processing photo) and 30% are expensive (i.e., processing video). In order to avoid infinite runs in the simulations, executions are set to stop at a given time limit. This allows us to observe the model's behavior up to a certain point in time. Figure 5.9 shows the number of failures to request calls (i.e., the missed deadlines) with respect to time and compares the performance of three different schedulers* fifo, edf, *and* sjf, *where* sjf *scheduler gives a better performance with respect to the other two schedulers.*

```
// A scheduler which switches strategy based on the length of the queue
def Process lengthsensitive(Int limit, List<Process> l) =
    if (length(l)<limit)
    then sjf(l)
    else fifo(l);

[Scheduler: lengthsensitive(limit,queue)]
class ServerImp (Int limit) implements Server {
  ... // (implementation unchanged)
}
```

Figure 5.10: The model from Fig. 5.8, with an application-specific scheduler which adapts to server load.

**Example 9.** *In the previous example, the* sjf *algorithm performed best overall, but this came at the cost of penalizing expensive jobs, who got a large portion of the total deadline misses. We now modify the model with an application-specific scheduler that can be parameterized to balance overall performance and fairness between large and small jobs.*

*Fig. 5.10 shows the modifications made to the model from Fig. 5.8. The new scheduler `lengthsensitive` switches between* sjf *and* fifo *behavior as the length of the process queue crosses a specified threshold. Note that the cutoff value `limit` is taken from the object state and is passed in as a second parameter to the scheduler. This shows how the state of an object can influence its scheduling decisions.*

*Switching between* fifo *and* sjf *represents a tradeoff between favoring short and long jobs in our usage scenario. The new `limit` parameter to the class `ServerImp` lets the modeler influence the ratio of deadline misses, and hence QoS, for each job type. Fig. 5.11 presents simulation results for varying cutoff points. We record results and show deadline misses as percentage of overall submitted jobs for large and small jobs separately. It can be seen that for `limit` $\leq 6$, the system performs identically to a system using* sjf *scheduling. For `limit` $\geq 15$, the behavior is the same as* fifo*. The `limit` parameter can thus be used to influence overall quality of service, and to dynamically adjust an object's behavior for changing workloads. Note that we simulate with a constant value for `limit`, but it is straightforward to, e.g., model a monitor object running in parallel with the server that adjusts this parameter at runtime to adjust scheduling behavior based on performance measuring.*

## 5.6 Conclusion

Whereas scheduling has traditionally been studied in the context of operating systems, modern software applications with soft real-time requirements need flexible

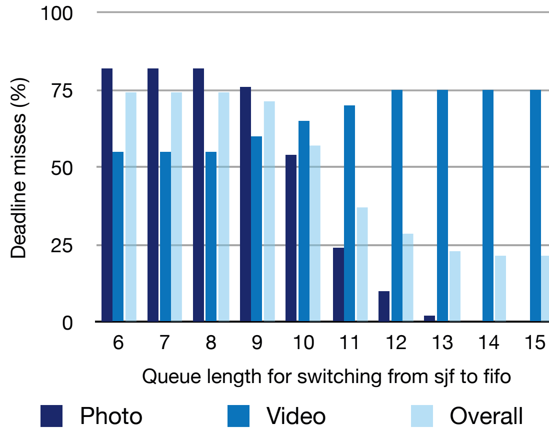


Figure 5.11: Application-specific scheduling for the server example: a conditional scheduler switches between *fifo* and *sjf* behavior, depending on object state and server load.

application-specific schedulers to control application-level performance. This paper has presented Real-Time ABS, a real-time object-oriented modeling language in which user-defined schedulers may be associated with concurrent objects and deadlines with method calls. We have defined a formal semantics for Real-Time ABS and shown how user-defined schedulers may be expressed at the abstraction level of the modeling language and integrated in the formal semantics. A tool based on an abstract interpreter has been implemented for Real-Time ABS, which can be used for simulation and measurements for Real-Time ABS models. A series of examples demonstrate modeling with user-defined schedulers in Real-Time ABS and the use of this tool. Recent complementary work has shown how schedulability analysis for Real-Time ABS can be done by means of an encoding into timed automata [10] and how cost estimates for methods in Real-Time ABS can be inferred using the COSTABS tool [2]. The integration of this work in our tool is currently underway.

# Bibliography


[1] Agha, G.A.: ACTORS: A Model of Concurrent Computations in Distributed Systems. The MIT Press, Cambridge, Mass. (1986)

[2] Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: Cost analysis of concurrent OO programs. In: Proc. 9th Asian Symposium on Programming Languages and Systems (APLAS 2011), *Lecture Notes in Computer Science*, vol. 7078, pp. 238–254. Springer (2011).

[3] Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: TIMES - a tool for modelling and implementation of embedded systems. In: Proc. 8th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002), *Lecture Notes in Computer Science*, vol. 2280, pp. 460–464. Springer (2002)

[4] Andrews, G.R.: Foundations of Multithreaded, Parallel, and Distributed Programming. Addison-Wesley (1999)

[5] Angerer, C.M., Gross, T.R.: Static analysis of dynamic schedules and its application to optimization of parallel programs. In: Proc. 23rd Languages and Compilers for Parallel Computing (LCPC 2010), *Lecture Notes in Computer Science*, vol. 6548, pp. 16–30. Springer (2010)

[6] Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (2007)

[7] Bjørk, J., Johnsen, E.B., Owe, O., Schlatte, R.: Lightweight time modeling in Timed Creol. *Proc. 1st Intl. Workshop on Rewriting Techniques for Real-Time Systems (RTRTS 2010).* Electronic Proceedings in Theoretical Computer Science **36**, 67–81 (2010).

[8] Blair, G.S., Coulson, G., Robin, P., Papathomas, M.: An architecture for next generation middleware. In: Proc. IFIP Intl. Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), pp. 191–206. Springer (1998)

[9] de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: R. de Nicola (ed.) Proc. 16th European Symposium on Programming (ESOP'07), *Lecture Notes in Computer Science*, vol. 4421, pp. 316–330. Springer (2007)

[10] de Boer, F.S., Jaghoori, M.M., Johnsen, E.B.: Dating concurrent objects: Real-time modeling and schedulability analysis. In: Proc. 21st Intl. Conf. on Concurrency Theory (CONCUR), *Lecture Notes in Computer Science*, vol. 6269, pp. 1–18. Springer (2010)

[11] Buttazzo, G.: Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, 2 edn. Kluwer Academic Publishers (2004)

[12] Chakravarti, A.J., Baumgartner, G., Lauria, M.: Application-specific scheduling for the organic grid. In: Proc. 5th IEEE/ACM Intl. Workshop on Grid Computing (GRID'04), pp. 146–155. IEEE Press (2004)

[13] Cheng, A.M.K.: Real-Time Systems: Scheduling, Analysis, and Verification. John Wiley & Sons, Inc. (2002)

[14] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, *Lecture Notes in Computer Science*, vol. 4350. Springer (2007)

[15] Coffman, E.G.: Computer and Job-Shop Scheduling Theory. John Wiley & Sons, Inc. (1976)

[16] David, A., Larsen, K.G., Legay, A., Nyman, U., Wasowski:, A.: ECDAR: An environment for compositional design and analysis of real time systems. In: Proc. 8th Automated Technology for Verification and Analysis (ATVA 2010), *Lecture Notes in Computer Science*, vol. 6252, pp. 365–370. Springer (2010)

[17] Dibble, P.C.: Real-Time Java Platform Programming, 2 edn. BookSurge Publishing (2008)

[18] Fersman, E., Krcál, P., Pettersson, P., Yi, W.: Task automata: Schedulability, decidability and undecidability. Information and Computation **205**(8), 1149–1172 (2007)

[19] Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theoretical Computer Science **410**(2–3), 202–220 (2009)

[20] Harchol-Balter, M., Schroeder, B., Bansal, N., Agrawal, M.: Size-based scheduling to improve web performance. ACM Transactions on Computer Systems **21**(2), 207–233 (2003)

[21] Hoare, C.A.R.: Monitors: an operating system structuring concept. Commun. ACM **17**, 549–557 (1974)

[22] Hsiung, P.A., Huang, C.H., Chen, Y.H.: Hardware task scheduling and placement in operating systems for dynamically reconfigurable soc. Journal of Embedded Computing **3**(1), 53–62 (2009)

[23] Hsu, C.H., Chen, S.C.: A two-level scheduling strategy for optimising communications of data parallel programs in clusters. Intl. Journal of Ad Hoc and Ubiquitous Computing **6**(4), 263–269 (2010)

[24] Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Proc. 9th Intl. Symposium on Formal Methods for Components and Objects (FMCO 2010), *Lecture Notes in Computer Science*, vol. 6957, pp. 142–164. Springer (2011).

[25] Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and Systems Modeling **6**(1), 35–58 (2007)

[26] Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer **1**(1–2), 134–152 (1997)

[27] Lee, E.A.: Computing needs time. Communications of the ACM **52**(5), 70–79 (2009)

[28] Logan, M., Merritt, E., Carlsson, R.: Erlang and OTP in Action. Manning Publications (2010)

[29] Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96**, 73–155 (1992)

[30] Nobakht, B., de Boer, F.S., Jaghoori, M.M., Schlatte, R.: Programming and deployment of active objects with application-level scheduling. In: Proc. Symposium on Applied Computing (SAC). ACM (2012).

[31] Ölveczky, P.C., Meseguer, J.: Specification of real-time and hybrid systems in rewriting logic. Theoretical Computer Science **285**(2), 359–405 (2002)

[32] Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation **20**(1–2), 161–196 (2007)

[33] Pierce, B.C.: Types and Programming Languages. The MIT Press (2002)

[34] Plotkin, G.D.: A structural approach to operational semantics. Journal of Logic and Algebraic Programming **60-61**, 17–139 (2004)

[35] Santoso, J., van Albada, G.D., Nazief, B.A.A., Sloot, P.M.A.: Simulating job scheduling for clusters of workstations. In: M. Bubak, H. Afsarmanesh, R. Williams, L.O. Hertzberger (eds.) 8th Intl. Conf. on High-Performance Computing and Networking (HPCN Europe 2000), *Lecture Notes in Computer Science*, vol. 1823, pp. 395–406. Springer (2000)

[36] Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing active objects to concurrent components. In: European Conf. on Object-Oriented Programming (ECOOP 2010), *Lecture Notes in Computer Science*, vol. 6183, pp. 275–299. Springer (2010)

[37] Schoeberl, M.: Real-time scheduling on a Java processor. In: Proc. 10th Intl. Conf. on Real-Time and Embedded Computing Systems and Applications (RTCSA) (2004)

[38] Sorensen, A., Gardner, H.: Programming with time: cyber-physical programming with Impromptu. In: W.R. Cook, S. Clarke, M.C. Rinard (eds.) Proc. Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA), pp. 822–834. ACM (2010)

[39] Srinivasan, S., Mycroft, A.: Kilim: Isolation-typed actors for Java. In: J. Vitek (ed.) Proc. 22nd European Conf. on Object-Oriented Programming (ECOOP 2008), *Lecture Notes in Computer Science*, vol. 5142, pp. 104–128. Springer (2008)

[40] Tchernykh, A., Ramírez-Alcaraz, J.M., Avetisyan, A., Kuzjurin, N., Grushin, D., Zhuk, S.: Two level job-scheduling strategies for a computational grid. In: R. Wyrzykowski, J. Dongarra, N. Meyer, J. Wasniewski (eds.) 6th Intl. Conf. on Parallel Processing and Applied Mathematics (PPAM), *Lecture Notes in Computer Science*, vol. 3911, pp. 774–781. Springer (2005)

CHAPTER 6

# Paper 2: Integrating Deployment Architectures and Resource Consumption in Timed Object-Oriented Models *

***Authors:*** Einar Broch Johnsen, Rudolf Schlatte and Silvia Lizeth Tapia Tarifa.

***Publication:*** Research Report 438, Department of Informatics, University of Oslo, February 2014. Submitted to the Journal of Logic and Algebraic Programming, in second round revision.

***Abstract:*** Software today is often developed for many deployment scenarios; the software may be adapted to sequential, concurrent, distributed, and even virtualized architectures. Since software performance can vary significantly depending on the target architecture, design decisions need to address which features to include and what performance to expect for different architectures. To make use of formal methods for these design decisions, system models need to range over deployment scenarios. For this purpose, it is desirable to lift aspects of low-level deployment to the abstraction level of the modeling language. This paper proposes an integration of deployment architectures in the Real-Time ABS language, with restrictions on processing resources. Real-Time ABS is a timed, abstract and behavioral specification language with a formal semantics and a Java-like syntax, that targets concurrent, distributed and object-oriented systems. A separation of concerns between execution cost at the object level and execution capacity at the deployment level makes it easy to

*This work was done in the context of the EU project FP7-610582 *ENVISAGE: Engineering Virtualized Services* (`http://www.envisage-project.eu`)

compare the timing and performance of different deployment scenarios already during modeling. The language and associated simulation tool is demonstrated on examples and its semantics is formalized.

**Keywords:** Deployment architecture; resources; real-time; object orientation; formal methods; performance; Real-Time ABS

## *6.1 Introduction*

Software is increasingly often developed as a range of systems. Different versions of a software may provide different functionality and advanced features, depending on the target users. A development method which attempts to systematize this software variability is product line engineering [1]; in a product line, different versions of a software (i.e., the products) may be instantiated with different features. An example is software for cell phones. Products for different cell phones and service subscriptions are produced by selecting among functional features such as call forwarding, answering machine, text messaging, etc. However, the selection of features in a product may be restricted by the hardware capacity of the different targeted cell phones. In addition to their functional variability, software systems need to adapt to different *deployment architectures*. For example, *operating systems* adapt to specific hardware and even to different numbers of available cores; *virtualized applications* are deployed on a varying number of (virtual) servers; and *services on the cloud* may need to dynamically adapt to the underlying cloud infrastructure and to changing load scenarios. This kind of adaptability raises new challenges for the modeling and analysis of component-based applications [2]. To apply formal methods to the design of such systems, it is interesting to lift aspects of low-level deployment concerns to the abstraction level of the modeling language.

The motivation for the work presented in this paper is to apply performance analysis to formal object-oriented models in which objects are deployed on resource-constrained deployment architectures. The idea underlying our approach is to make a separation of concerns between the *cost* of performing a computation and the available resource *capacity* of the deployment architecture, rather than to assume that this relationship is fixed in terms of, e.g., specified execution times. Although a range of resources could be considered, this paper focuses on processing capacity. In our approach, the underlying deployment architecture of the targeted system forms an integral part of the system model, but defaults are provided which allow the modeler to ignore architectural design decisions when desirable. The separation of concerns between cost and capacity allows the performance of a model to be compared for a range of deployment choices. By comparing deployment choices many interesting questions concerning performance can be addressed during the system design phase,

for example:

- How will the response time of my system improve if I double the number of servers?
- How do fluctuations in client traffic influence the performance of my system on a given deployment architecture?
- Can I better control the performance of my system by means of application-specific load balancing?

Our approach is based on ABS [3], a modeling language for distributed concurrent object groups akin to concurrent objects (e.g., [4–6]), Actors (e.g., [7,8]), and Erlang processes [9]. Concurrent object groups communicate by asynchronous method calls and futures [6]. ABS is an executable imperative language which allows modeling abstractions; for example, functions and algebraic data types can be used to abstract from imperative data structures while retaining an overall object-oriented design. ABS has a formal semantics in an SOS style [10] as well as a tool suite to support the development and analysis of models [11]. The core of this tool suite is an editor in Eclipse with a compiler and a language interpreter executing on Maude [12], a platform for programs written in rewriting logic [13]. The syntax of ABS and its semantics for sequential object-oriented programs are sufficiently similar to industrial programming languages (specifically, Java) that a moderately experienced software engineer can start using it with a reasonably small learning effort.

To model object-oriented applications in resource-constrained deployment architectures, we extend ABS with *deployment components*. Deployment components were originally proposed by the authors in [14]. Deployment components capture the execution capacity of a location in the deployment architecture, on which a number of concurrent objects are deployed. Deployment components are parametric in the amount of concurrent execution capacity they allow within a time interval. This allows us to analyze how the execution capacity of a deployment component influences the performance of objects executing on the deployment component. The authors also extended this approach to support dynamic resource reallocation [15] and object mobility [16]. This paper improves and combines results from [14–16] by, first, giving a *unified* presentation of this work; second, adapting our approach to a dense *real-time* model whereas the previous papers used discrete time; third, refining the cost model of the previous papers [14,15] from fixed costs to the *flexible user-defined cost expressions* introduced in [16]; and fourth, refining the semantics to directly handle slow computations which require several time intervals. To validate and compare the concurrent behavior of models under restricted concurrency assumptions, we use the tool suite for Real-Time ABS.

**Paper overview.** Section 6.2 presents the Real-Time ABS modeling language and Section 6.3 extends Real-Time ABS with a deployment layer to capture deployment

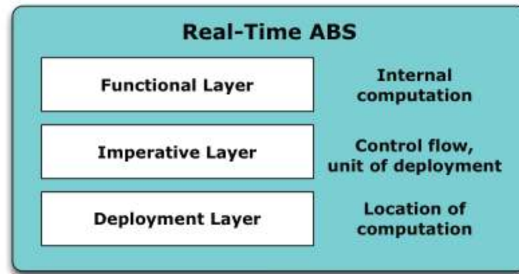


Figure 6.1: The Layers of the Real-Time ABS Modeling Language

architectures and resource consumption. Sections 6.4, 6.5, and 6.6 highlight different aspects of the modeling language through examples and show how the Real-Time ABS tool can be used to obtain insights into the deployment aspects of the models. Section 6.7 formalizes the modeling language in terms of an operational semantics. Section 6.8 discusses related and future work, and Section 6.9 concludes the paper.

## 6.2 Modeling Timed Behavior in Real-Time ABS

ABS is an executable object-oriented modeling language which combines functional and imperative programming styles to develop high-level executable models. ABS targets the modeling of distributed systems by means of concurrent object groups that internally support interleaved concurrency. Concurrent object groups execute in parallel and communicate through asynchronous method calls. A concurrent object group has at most one active process at any time and a queue of suspended processes waiting to execute on an object in the group. This makes it very easy to combine active and reactive behavior in the concurrent object groups, based on a cooperative scheduling [3] of processes which stem from method activations. Objects in ABS are dynamically created from classes typed by interface; i.e., there is no explicit notion of hiding as the object state is always encapsulated behind interfaces which offer methods to the environment.

Inside an object, internal computation is captured in a simple functional language based on user-defined algebraic data types and functions. Thus, the modeler may abstract from many details of the low-level imperative implementations of data structures, and still maintain an overall object-oriented design which is close to the target system. A schematic view of the modeling layers of ABS is given in Figure 6.1; this section presents the functional and imperative layers, the deployment layer is discussed

| *Syntactic categories.* | *Definitions.* |
|---|---|
| $T$ in GroundType | $T ::= B \mid I \mid D \mid D\langle\overline{T}\rangle$ |
| $A$ in Type | $A ::= N \mid T \mid N\langle\overline{A}\rangle$ |
| $x$ in Variable | $Dd ::= \textbf{data}\ D[\langle\overline{A}\rangle] = [\overline{Cons}];$ |
| $e$ in Expression | $Cons ::= Co[(\overline{A})]$ |
| $v$ in Value | $F ::= \textbf{def}\ A\ fn[\langle\overline{A}\rangle](\overline{A}\ \overline{x}) = e;$ |
| $br$ in Branch | $e ::= x \mid v \mid Co[(\overline{e})] \mid fn(\overline{e}) \mid \textbf{case}\ e\ \{\overline{br}\}$ |
| $p$ in Pattern | $\mid \textbf{this} \mid \textbf{now}() \mid \textbf{deadline}() \mid \textbf{destiny}()$ |
| | $v ::= Co[(\overline{v})] \mid \textbf{null}$ |
| | $br ::= p \Rightarrow e;$ |
| | $p ::= _ \mid x \mid v \mid Co[(\overline{p})]$ |

Figure 6.2: Syntax for the functional layer of Real-Time ABS. Terms $\overline{e}$ and $\overline{x}$ denote possibly empty lists over the corresponding syntactic categories, and square brackets [ ] optional elements.

in Section 6.3.

At a high level of abstraction, concurrent object groups typically consist of a single concurrent object; other objects may be introduced into a group as required to give some of the algebraic data structures an explicit imperative representation when this is natural in a model. To simplify the presentation in this paper, we aim at high-level models and only consider concurrent objects (i.e., the groups will always consist of single concurrent objects). We make use of ABS *annotations* (a general mechanism to add meta-data to statements) to express timing and deployment aspects in our models. This paper assumes that all ABS programs are well-typed. In particular, the presented syntax definition and operational semantics assume that annotations only occur as discussed in the sequel. (A type system for the core ABS language is given in [3].)

Real-Time ABS [17] is an extension of ABS to model the timed behavior of concurrent objects in ABS. The object-oriented perspective on timed behavior is captured by *deadlines* on method calls. Every method activation in Real-Time ABS has an associated deadline; this deadline captures the remaining execution time, so it decreases with the passage of time. Deadlines are *soft*; i.e., the execution of the method does not stop because the deadline is missed. By default the deadline associated with a method activation is infinite, so in an untimed ABS model deadlines will never be missed. We use the annotation mechanism of ABS to override the default deadline for specific method calls.

### 6.2.1 The Functional Layer of Real-Time ABS

The functional layer of Real-Time ABS consists of a library of algebraic data types such as the empty type Unit, booleans Bool, integers Int, rational numbers Rat, and

strings String; parametric data types such as sets Set⟨A⟩ and maps Map⟨A, B⟩ (given values for the type variables A and B); and (parametric) functions over values of these data types. For simplicity, Real-Time ABS does not support operator overloading.

**Example 10.** *Polymorphic sets in Real-Time ABS. Polymorphic sets can be defined using a type variable A and two constructors EmptySet and Insert. We define a function contains which recursively checks whether an element el is in a set ss by pattern matching over ss.*

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

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

*The underscore _ matches any element in a constructor pattern without introducing a variable binding, the new binding xs matches the rest of the set in the last* **case** *clause.*

*The following statement defines a variable b and sets its value to True.*

```
  Bool b = contains(1, Insert(1, EmptySet));
```

To express time, we consider a dense time model represented by two types Time and Duration. Time values capture points in time as reflected on a global clock during execution. In contrast, finite durations reflect the execution time (i.e., the difference between two time values). However, durations may be infinite (or unbounded). Infinite durations are captured by the term InfDuration, which is such that for all other durations $d_1, d_2$, the sum $d_1 + d_2$ is smaller than InfDuration.

**Example 11.** *Dense Time in Real-Time ABS. The datatypes Time and Duration are defined as follows:*

```
data Time = Time(Rat timeValue);

data Duration = Duration(Rat durationValue) | InfDuration;
```

*Here, timeValue and durationValue are partially defined* accessor *functions on the types Time and Duration. For example, given a rational number r, the value Time(r) is of type Time, Duration(r) is of type Duration, and the expressions timeValue(Time(r)) and durationValue(Duration(r)) both evaluate to r. Functions are defined in a standard way, shown here are a subtraction function timeDifference on Time values, a unary predicate `isInfinite` on Duration values, and the binary less-than relation `lt` on Duration values:*

```
def Duration timeDifference(Time t1, Time t2) =
  Duration(timeValue(t1)−timeValue(t2));

def Bool isInfinite(Duration d) = d == InfDuration;

def Bool lt(Duration d1, Duration d2) =
  case d1 { InfDuration => False;
    Duration(v1) => case d2 {
      InfDuration => True;
      Duration(v2) => v1 < v2;};};
```

*Two* Duration *values can be added. Since there is no operator overloading in Real-Time ABS, we define addition of durations as a function* `add`*:*

```
def Duration add(Duration d1, Duration d2) =
  case d1 { InfDuration => InfDuration;
    Duration(v1) => case d2 {
      InfDuration => InfDuration;
      Duration(v2) => Duration(v1 + v2);};};
```

*Here, the operators* $<$ *and* + *are used for comparison and addition in the underlying datatype of rational numbers.*

The formal syntax of the functional language is given in Figure 6.2. The ground types $T$ consist of basic types $B$ such as Bool and Int, as well as names $D$ for datatypes and $I$ for interfaces. In general, a type $A$ may also contain type variables $N$ (i.e., uninterpreted type names [18]). In *datatype declarations* $Dd$, a datatype $D$ has a set of constructors $Cons$, each of which has a name $Co$ and a list of types $\overline{A}$ for their arguments. *Function declarations* $F$ have a return type $A$, a function name $fn$, a list of parameters $\overline{x}$ of types $\overline{A}$, and a function body $e$. Both datatypes and functions may be polymorphic and have a bracketed list of type parameters (e.g., Set⟨Bool⟩). The layered type system allows functions in the functional layer to be defined over types $A$ which are parametrized by type variables but only applied to ground types $T$ in the imperative layer; e.g., the head of a list is defined for List⟨A⟩ but applied to ground types such as List⟨Int⟩.

*Expressions* $e$ include variables $x$, values $v$, constructor expressions $Co(\overline{e})$, function expressions $fn(\overline{e})$, case expressions **case** $e$ $\{\overline{br}\}$, the self-identifier **this**, **now**() of type Time, which evaluates to the current value of the global system clock, **deadline**() of type Duration, which evaluates to the remaining execution time before the reply from the current process is due, and **destiny**() which refers to the future variable where the return value from the current process is stored after finishing the execution (see the next section). *Values* $v$ are expressions which have reached a normal form; i.e., constructors applied to values $Co(\overline{v})$ or **null** (omitted from Figure 6.2 are values of the basic types String, Rat and Int, which are standard).

*Case expressions* match a value against a list of branches $p \Rightarrow e$, where $p$ is a

pattern. Patterns are composed of the following elements:

- wild cards _ which match anything;
- variables $x$ match anything if they are free or match against the existing value of $x$ if they are bound;
- values $v$ which are compared literally;
- constructor patterns $Co(\overline{p})$ which match $Co$ and then recursively match the elements $\overline{p}$.

The branches are evaluated in the listed order, free variables in p are bound in the expression e.

### 6.2.2 The Imperative Layer of Real-Time ABS

The imperative layer of Real-Time ABS addresses concurrency, communication, and synchronization in the system design, and defines interfaces, classes, and methods in an object-oriented language with a Java-like syntax. In Real-Time ABS, concurrent objects are *active* in the sense that their run method, if defined, gets called upon creation.

*Statements* are standard for sequential composition $s_1; s_2$, and for **skip**, **if**, **while**, and **return** constructs. The statement **duration**$(e_1, e_2)$ causes time to advance between a best case $e_1$ and a worst case $e_2$ execution time, where $e_1$ and $e_2$ are rational numbers. Cooperative scheduling in ABS is achieved by explicitly suspending the execution of the active process. The statement **suspend** unconditionally suspends the execution of the active process and moves this process to the queue. The statement **await** $g$ conditionally suspends execution; the guard $g$ controls processor release and consists of Boolean conditions $b$ and return tests $x$? (explained in the next paragraph). Just like expressions $e$, the evaluation of guards $g$ is side-effect free. However, if $g$ evaluates to false, the processor is released and the process *suspended*. When the execution thread is idle, an enabled task may be selected from the pool of suspended tasks by means of a default scheduling policy. In addition to expressions $e$, the right hand side of an assignment $x{=}rhs$ includes object group creation **new cog** $C(\overline{e})$, method calls $o!m(\overline{e})$, and future dereferencing $x$.**get**. Method calls and future dereferencing are explained in the next paragraph.

*Communication* and *synchronization* are decoupled in Real-Time ABS. Communication is based on asynchronous method calls, denoted by assignments $f{=}o!m(\overline{e})$ to future variables $f$ of type **Fut**$\langle T\rangle$, where $T$ corresponds to the return type of the called method $m$. Here, $o$ is an object expression, $m$ a method name, and $\overline{e}$ are expressions providing actual parameter values for the method invocation. (Local calls are

written **this**!$m(\overline{e})$.) After calling $f$=$o$!$m(\overline{e})$, the future variable $f$ refers to the return value of the call, and the caller may proceed with its execution *without blocking*. Two operations on future variables control synchronization in Real-Time ABS. First, the guard **await** $f$? *suspends the active process* unless a return to the call associated with $f$ has arrived, allowing other processes in the object to execute. Second, the return value is retrieved by the expression $f$.**get**, which *blocks all execution* in the object until the return value is available. Futures are first-class citizens of Real-Time ABS; the expression **destiny**() refers to the future associated with the current process [6]. The statement sequence $x$=$o$!$m(e)$;$v$=$x$.**get** encodes commonly used *blocking calls*, abbreviated $v$=$o.m(e)$ (often referred to as synchronous calls). If the return value of a call is without interest, the call may occur directly as a statement $o$!$m(e)$ with no associated future variable. This corresponds to asynchronous message passing. The default deadline of a method activation is InfDuration. However, this default may be overridden by an optional *deadline annotation* to the method call statement, which takes as its argument a duration value. Note that deadline annotations can only occur associated with method calls.

**Example 12.** *Deadlines. Assume that an object* o *implements a method* m *which takes a formal parameter of type $T$. We define a wrapper method* n *which calls* m *on* o *and specify a deadline for this synchronized call, given as an annotation and expressed in terms of its own remaining deadline. The method n succeeds if it can return within its given deadline. Note that if its own deadline is* InfDuration*, then the deadline to* m *will also be unlimited. The function* scale(d,r) *multiplies the value of a duration* d *by a rational number* r *(we omit the definition of* scale*, which is straightforward):*

```
Unit n (T x){
  [Deadline: scale(deadline(), 9/10)] f=o.m(x);
  return deadline() > 0;
}
```

The formal syntax of the imperative layer of Real-Time ABS is given in Figure 6.3. A program $P$ consists of lists of interface and class declarations followed by a main block $\{\overline{T}\ \overline{x}; s\}$, which is similar to a method body. An interface $IF$ has a name $I$ and method signatures $Sg$. A class $CL$ has a name $C$, interfaces $\overline{I}$ (specifying types for its instances), class parameters and state variables $x$ of type $T$, and methods $M$ (The *attributes* of the class are both its parameters and state variables). A method signature $Sg$ declares the return type $T$ of a method with name $m$ and formal parameters $\overline{x}$ of types $\overline{T}$. $M$ defines a method with signature $Sg$, local variable declarations $\overline{x}$ of types $\overline{T}$, and a statement $s$. Statements may access attributes, locally defined variables, and the method's formal parameters. There are no type variables at the imperative layer of Real-Time ABS.

*Syntactic categories.*
$s$ in Stmt
$e$ in Expr
$b$ in BoolExpr
$a$ in Annotation
$g$ in Guard

*Definitions.*

$$
\begin{aligned}
P &::= \overline{IF}\ \overline{CL}\ \{[\overline{T}\ \overline{x};]\ s\,\}\\
IF &::= \textbf{interface}\ I\,\{\,[\overline{Sg}]\,\}\\
CL &::= \textbf{class}\ C\,[(\overline{T}\ \overline{x})]\,[\textbf{implements}\,\overline{I}]\,\{\,[\overline{T}\ \overline{x};]\,\overline{M}\}\\
Sg &::= T\ m\ ([\overline{T}\ \overline{x}])\\
M &::= Sg\ \{[\overline{T}\ \overline{x};]\ s\,\}\\
s &::= s; s \mid [\,[a]\,]\,s \mid \textbf{skip} \mid x = rhs \mid \textbf{if}\ b\ \{\,s\,\}\,[\textbf{else}\,\{\,s\,\}]\\
&\quad\mid \textbf{while}\ b\,\{\,s\,\} \mid \textbf{duration}(e, e) \mid \textbf{suspend}\\
&\quad\mid \textbf{await}\ g \mid \textbf{return}\ e\\
a &::= \texttt{Deadline:}\ e\\
rhs &::= e \mid cm \mid \textbf{new cog}\ C\,(\overline{e})\\
cm &::= e!m(\overline{e}) \mid x.\textbf{get}\\
g &::= b \mid x? \mid g \wedge g
\end{aligned}
$$

Figure 6.3: Syntax for the imperative layer of Real-Time ABS. Terms like $\overline{e}$ and $\overline{x}$ denote (possibly empty) lists over the corresponding syntactic categories, square brackets [ ] denote optional elements.

### 6.2.3 Explicit and Implicit Time in Real-Time ABS

In Real-Time ABS, the local passage of time can be modeled both *explicitly* and *implicitly*. With explicit time, the modeler inserts duration statements with best-case and worst-case execution times into the model. This is the standard approach to modeling timed behavior, well-known from, e.g., timed automata in UPPAAL [19]. Duration statements specify explicit execution times when the model abstracts from the system's deployment architecture (e.g., the deployment architecture is assumed to be fixed and the load captured by worst- and best-case execution times).

**Example 13.** *Explicit time. Let f be a function defined in the functional layer of Real-Time ABS, which recurses through some data structure x of type T, and let the function size measure this data structure. Consider a method m which takes as input such a value x and returns the result of applying f to x. Let us assume that the time needed for this computation depends on the size of x; e.g., the execution time is between a duration size(x)/2 and a duration 4∗size(x). An interface I which provides the method m and a class C which implements I, including the execution time for m using the explicit time model, are specified as follows:*

```
interface I {
  Int m(T x)
}

class C implements I {
  Int m (T x){
     duration(size(x)/2, 4*size(x));
     return f(x);
  }
}
```

With implicit time the execution time is not specified explicitly in terms of durations, but rather *observed* on the executing model. This is done by comparing clock values from the global clock during model execution.

**Example 14.** *Implicit time. We specify an interface* J *with a method* p *which, given a value of type* T*, returns a value of type* Duration*, and we implement* p *in a class* D *such that* p *measures the time needed to call the method* m *of Example 13 above, as follows:*

```
interface J {
    Duration p (T x)
}

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

Observe that with implicit time, no assumptions about execution times are given in the model. The execution time depends on how quickly the method call is effectuated by the other object. In Example 14 the execution time is simply observed by comparing the time before and after making the call. As a consequence, the time needed to execute a statement depends on the *capacity* of the chosen deployment architecture and on *synchronization* with (slower) objects. In many cases it is natural to use both explicit and implicit time in a model, so both are supported in Real-Time ABS.

## 6.3 Modeling Deployment Architectures

The execution time in a distributed system depends on the amount of computation which takes place at different locations, and on the execution capacity of those locations. Deployment architectures express how different software units are deployed on physical or virtual hardware. We can think of a deployment architecture as a collection of locations with resource constraints, on which the software units are deployed. Real-Time ABS makes a separation of concerns between the resource *cost* of performing a computation and the resource *capacity* of a given location.

In this paper we focus on CPU resources. Deployment components are used to model locations which are restricted in their execution capacity. Resource cost annotations are used to express resource consumption during computation. Other resource

```
data DCData = InfCPU | CPU(Int capacity);

interface DC{
  DCData total();
  Rat load(Int n);
  Unit transfer(DC target, Int amount)
}
```

Figure 6.4: The specification of resources and the interface of deployment components.

types like bandwidth, memory, and power consumption can be handled with similar techniques as the ones presented here; work in this area is ongoing.

### 6.3.1 Deployment Components

A *deployment component* captures the execution capacity of a location on which a number of concurrent objects are deployed. The capacity is specified as an amount of resources which is available during a time interval; the time interval corresponds to the time between integer values in the dense time domain of Real-Time ABS. The available resources may be used to perform computation within the time interval, and they are renewed for the next time interval. The renewal of resources for different deployment components is synchronized.

The main block of the model executes in a root object located on a default deployment component, which we call environment, with unrestricted processing capacity. A model may be extended with other deployment components with different capacities. When objects are created, they are by default allocated to the same deployment component as their creator, but they may also be allocated to a different deployment component. Thus, in a model without explicit deployment components all objects run in environment, which places no restrictions on the processing capacity of the model.

Deployment components are first-class citizens of Real-Time ABS. They may be passed around as arguments to method calls, they support a number of methods, and they may be created dynamically, depending on control flow, or statically in the main block of the model. Syntactically, deployment components in Real-Time ABS are manipulated in a way similar to objects. Variables which refer to deployment components are typed by an interface DC and new deployment components are dynamically created as instances of a class DeploymentComponent, which implements DC. The interface DC is defined as in Figure 6.4.

The DC interface provides the following methods for resource management: **total**() returns the number of resources currently allocated to the deployment component, **load**(n) returns the deployment component's average load during the last n time intervals in a percentage scaled from 0 to 100 (i.e., a load of 90 means that 90% of the total

resources have been used in the last $n$ time intervals in average), and **transfer**(target,r) reallocates r resources from the deployment component to the target deployment component. If the deployment component has less than r resources available, the available amount is transferred.

The type DCData, given in Figure 6.4, reflects processing capacity. It has constructors InfCPU for infinite resources and CPU(r), where r represents the amount of processing resources available in a time interval. The observer function capacity is defined for the constructor CPU(r) and returns the amount r. Deployment components are created by an assignment with the right hand side **new cog** DeploymentComponent(descriptor, capacity). The parameter capacity of type DCData specifies the initial CPU capacity of the deployment component. The parameter descriptor of type String is a descriptor mainly used for monitoring purposes; i.e., it defines a user-defined name for the deployment component which facilitates querying the run-time state but that has no semantic effect. The use of descriptors is further illustrated in the examples of Sections 6.4 and 6.6. Objects are deployed on deployment components when the objects are created. By default an object is deployed on the same deployment component as its creator. However, a different deployment component may be selected by means of an optional *deployment annotation* [DC: e] to the object creation statement, where e is an expression of type DC. Note that deployment annotations can only occur associated with the creation of concurrent object groups.

**Example 15.** *Static Deployment Architecture. Given the interfaces I and J and classes C and D of Examples 13 and 14, we can specify a static deployment architecture in which two C objects, deployed on different deployment components Server1 and Server2, interact with D objects deployed on a deployment component ClientServer, as follows. We create three deployment components with descriptors Server1, Server2, and ClientServer and processing capacities 6, 3, and InfCPU (i.e., the ClientServer has no resource restrictions). The local variables dc1, dc2, and dc3 refer to these three deployment components in the scope of the main block of the model. Objects are explicitly allocated to the servers by deployment annotations; below, object1 is allocated to Server1, etc.*

```
{
  // This main block initializes a static deployment architecture:
  DC dc1 = new cog DeploymentComponent("Server1",CPU(6));
  DC dc2 = new cog DeploymentComponent("Server2",CPU(3));
  DC dc3 = new cog DeploymentComponent("ClientServer", InfCPU);
  [DC: dc1] I object1 = new cog C;
  [DC: dc2] I object2 = new cog C;
  [DC: dc3] J client1monitor = new cog D(object1);
  [DC: dc3] J client2monitor = new cog D(object2);
}
```

*Figure 6.5 depicts this deployment architecture and the artefacts introduced into the modeling language.*

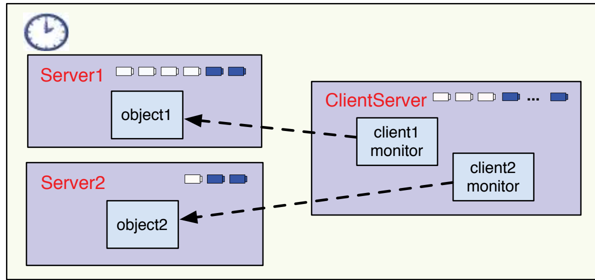


Figure 6.5: A deployment architecture in Real-Time ABS, with the three deployment components Server1, Server2, and ClientServer described in Section 6.3.1. In each deployment component, we see its allocated objects and the "battery" of allocated and available processing resources (top right).

### 6.3.2 Resource Consumption

The available resource capacity of a deployment component determines how much computation may occur in the objects deployed on that deployment component. Objects allocated to the deployment component compete for the shared resources in order to execute, and they may execute until the deployment component runs out of resources or they are otherwise blocked. In the case of CPU resources, the resources of the deployment component define its capacity inside a time interval, after which the resources are renewed.

The resource consumption of executing statements in the ABS model is determined by a default cost value which can be set as a compiler option (e.g., −defaultcost=10). However, the default cost does not discriminate between the statements, so a more refined cost model will often be desirable. For example, in a realistic model the assignment x=e should have a significantly higher cost for a complex expression e than for a constant. For this reason, more fine-grained costs can be inserted into Real-Time ABS models by means of *cost annotations* [Cost: e]. Note that cost annotations can be associated with any statement, and that a statement may have several annotations (for example, a **new** statement may have both a cost and a DC annotation; see Figures 6.3 and 6.6).

**Example 16.** *Annotation with concrete cost. Reconsider the class* C *of Example 13 and assume that the exact cost of computing the function* f(x) *may be given as a function* g *which depends on the size of the input value* x*. In the context of deployment components, a* resource-sensitive *implementation of interface* I *may be modeled, which does not have a predefined duration as in the explicit time model of class* C*. The resulting class* C2 *can be defined as follows:*

*Definitions.*

$a ::= \texttt{DC:}\ e \mid \texttt{Cost:}\ e \mid a, a \mid \ldots$

$e ::= \textbf{thisDC}(\,) \mid \ldots$

$rhs ::= \textbf{new cog}\ \texttt{DeploymentComponent}\,(e, e) \mid \ldots$

$cm ::= e!\textbf{load}(e) \mid e!\textbf{total}(\,) \mid e!\textbf{transfer}(e, e) \mid \ldots$

$s ::= \textbf{movecogto}(e) \mid \ldots$

Figure 6.6: Syntax extension for the deployment layer.

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

It is the responsibility of the modeler to specify appropriate resource costs. A behavioral model with default costs may be gradually refined to provide more realistic resource-sensitive behavior. For the computation of the cost functions such as g in Example 16, the modeler may be assisted by the COSTABS tool [20], which can compute a worst-case approximation of the cost of f in terms of abstract execution steps for an input value x based on static analysis techniques, when given the ABS definition of the expression f. However, the modeler may also want to capture resource consumption at a more abstract level; for example, resource limitations can be made explicit in the model during the early stages of system design. Therefore, cost annotations may be used by the modeler to abstractly represent the cost of some computation which is not fully specified.

**Example 17.** *Annotation with abstract cost. The class* C3 *below may represent a draft version of our method* m *from Example 16, in which the cost of the computation is specified although the function* f *has yet to be introduced:*

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

### *6.3.3 The Deployment Layer of Real-Time ABS*

Figure 6.6 summarizes the extensions to the syntax (as presented in Figures 6.2 and 6.3) for modeling deployment in Real-Time ABS. Annotations $a$ are extended with deployment component annotations [DC: e] as explained in Section 6.3.1) and cost annotations [Cost: e] as explained in Section 6.3.2. Expressions $e$ are extended with

**thisDC**(); since all objects are deployed on some deployment component, we let the expression **thisDC**() refer to the deployment component where the object is currently deployed, similar to the self reference **this**. The right hand side *rhs* of assignments is extended with deployment component creation **new cog** DeploymentComponent(descriptor, capacity) explained in Section 6.3.1. Method invocation *cm* is extended with methods **load**(n), **total**(), and **transfer**(target,amount), also explained in Section 6.3.1. Statements *s* are extended with a primitive **movecogto**(e) for object group reallocation, an object may relocate its concurrent object group to a deployment component e by executing this statement.

## *6.4 Example: A Client-Server System*

This section presents the first of three larger examples. We illustrate the modeling of deployment architecture and resource consumption through a client-server system and its behavior under various constant load scenarios. To focus on the mechanisms of the modeling of deployment architecture and resource consumption, we consider a simple model of a client-server system that models the general architecture and control flow of, e.g., a website or computation service, while mostly abstracting from the internal software architecture of the concrete system.

On the server, an agent distributes sessions to clients from a pool of session objects and dynamically creates new session objects as required (at a somewhat higher cost than re-using existing sessions). A client obtains a session through the getSession method of the Agent object; the session objects return themselves to the agent when the session is completed. Clients submit work to the server by calling the order method of a Session object, with a cost parameter that allows the model to specify the execution costs of the invoked service while abstracting from the concrete implementation of the service. Each session stays valid for one order, after which the client can ask for a new session. The Real-Time ABS model of the server is given in Figure 6.7.

In the implementation of the Session class the completion of an order requires a specific amount of resources, specified via its cost parameter. The **skip** statement in the order method consumes the given cost. An order is successful if it is completed within its deadline; success is calculated by checking that the **deadline**() expression is larger than Duration(0). Note that when sessions run on a deployment component with unlimited resources InfCPU, all orders will be completed immediately, as expected from an infinitely fast server. In the Agent class, the attribute sessions stores a set of Session objects which are currently not in use by any client (the ABS datatype for sets has two constructors EmptySet and Insert, and operations such as, emptySet to check for the empty set, take to select some element of a non-empty set, and remove to remove an element from a set). When a client requests a Session, the Agent takes a session from the set of available sessions if possible, otherwise it creates a new session.

```
interface Agent {
  Session getSession();
  Unit free(Session session, Bool success);
}

interface Session {
  Bool order(Int cost);
}

class Session(Agent agent) implements Session {
  Bool order(Int cost) {
    [Cost: cost] skip;
    Bool success = durationValue(deadline()) > 0;
    agent!free(this, success);
    return success;
  }
}

class Agent implements Agent {
  Set<Session> sessions = EmptySet;
  Int requestcount = 0;
  Int successcount = 0;

  Session getSession() {
    Session session;
    if (emptySet(sessions)) {
      [Cost: 2]session = new cog Session(this);
    }
    else {
      [Cost: 1]session = take(sessions);
      sessions = remove(sessions, session);
    }
    requestcount = requestcount + 1;
    return session;
  }

  Unit free(Session session, Bool success) {
    if (success) {
      successcount = successcount + 1;
    }
    sessions = Insert(session, sessions);
  }
}
```

Figure 6.7: A session-oriented server model in Real-Time ABS. An Agent object hands out Session objects, reusing them if possible. The behavior of the order method itself is left abstract.

Both re-using a session object and creating a new session have associated costs, to accurately model behavior under heavy load or denial-of-service attacks from the environment. The method free inserts a session in the available sessions of the Agent, and is called by the session itself upon completion of an order.

```
interface Client {}

class Client (Agent agent, Int cycle, Int cost, Int deadline) implements Client {
  Int ordercount = 0;
  Int successcount = 0;

  Unit run() {
    await duration(cycle, cycle);
    Session session = agent.getSession();
    [Deadline: Duration(deadline)] Fut<Bool> f = session!order(cost);
    ordercount = ordercount + 1;
    this!run();
    await f?;
    Bool result = f.get;
    if (result) {
       successcount = successcount + 1;
    }
  }
}

{
  //Main block
  DC shop = new cog DeploymentComponent("Shop", CPU(20));
  [DC: shop] Agent agent = new cog Agent();
  Client client1 = new cog Client(agent, 2, 5, 5);
  ...
}
```

Figure 6.8: Deployment environment and client model of the web shop example.

***Simulating and Testing the Server*** The behavior of the server can be analyzed by extending the model with a deployment scenario and an environment to simulate a workload. The operational semantics of Real-Time ABS with deployment components and resource consumption, presented in Section 6.7, has been specified in rewriting logic [13], which allows models to be analyzed using the rewriting tool Maude [12]. Given an initial configuration, Maude supports simulation and breadth-first search through reachable states to check safety properties and model checking of finite reachable states for LTL properties. In this paper, Maude is used as an interpreter for the semantics of Real-Time ABS to simulate and test Real-Time ABS models with deployment components and resource consumption.

The environment is modeled by creating one or more instances of the class Client, given in Figure 6.8. An instance of Client periodically calls order every c time intervals, corresponding to *periodic requests*. (The work in [14] showed the effects of clients with varying co-operative vs. flooding behavior on a similar model.) In the main block of the model, shown in Figure 6.8, a deployment component shop is created with a processing capacity of 20 resources available for the objects allocated on the shop. An instance of Agent is created in that deployment component, which in turn creates

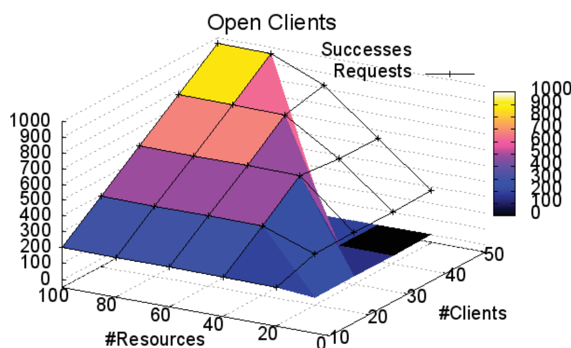


Figure 6.9: Number of total requests and successful orders, depending on the number of clients and resources. Once the load increases over a certain threshold, no deadlines are met in the simulated system.

Session objects when required by clients.

Figure 6.9 shows the number of total requests and successful orders for a set of simulation runs, where each run lasted for a duration of 100 time intervals. The scenarios range from 10 to 50 clients and from 20 to 100 resources on the shop deployment component. The scenario shows the effect of flooding the server with requests: After a certain threshold of incoming requests, server response QoS (expressed in responses within deadline vs. requests) collapses and no requests are successfully processed within the specified deadline.

## 6.5 Example: Implementing Object Migration to Mitigate Overload

In this section, the example of Section 6.4 is extended to dynamic deployment scenarios based on the migration of concurrent object groups. Figure 6.9 showed how heavy client traffic may lead to congestion on the server, which in turn can cause serious degradation of the server's quality of service. In order to investigate the effects of more dynamic deployment scenarios on the quality of service of timed software models, we compare the behavior of Real-Time ABS models with the same functional behavior and workload when the models are run on two different dynamic deployment scenarios.

Real-Time ABS models can include *load balancing strategies*, which aim to decrease congestion and thus improve the overall quality of service compared to models with static deployment scenarios. Load balancing strategies are typically expressed in Real-

```
interface Session { ...
  Unit moveTo(DC dc);
}

class SessionImp(Agent agent) implements Session {
  ...
  Unit moveTo(DC dc) {
    if (dc != thisDC()) {
      [Cost: 1] movecogto(dc);
      [Cost: 1] skip;
    }
  }
}

class SmartAgent(DC backupserver) implements Agent {
  ...
  Unit free(Session session) {
    ...
    session!moveTo(thisDC());
  }

  Session getsession() {
    ...
    Rat load = thisDC().load(1);
    DCData total = thisDC().total();
    if (total != InfCPU && load > 50) {
      session!moveTo(backupserver);
    }
    return session;
  }
}
```

Figure 6.10: An agent which performs load balancing. If the main server load is more than 50%, sessions are started on the backup server. (Code which is identical to that of Figure 6.8 has been elided for brevity.)

Time ABS using the resource-related language constructs **total** and **load** to inspect the state of the deployment architecture. For our server example, two sensible load balancing strategies might be to start requests on a *backup server* once the main server's load exceeds a certain threshold, or to migrate long-running requests to the backup server in order to free resources on the main server. This can be done by moving concurrent object groups between two deployment components, using the **movecogto** primitive.

In this section we model and simulate these two different load balancing strategies: (1) a *load balancing agent* which starts sessions on a backup server when the load on the main server is above a given threshold and (2) *self-monitoring sessions* which move themselves to the backup server once the processing of their current request exceeds a given time limit. Both of these dynamic deployment scenarios are analyzed

```
class SmartSession(Agent agent, Duration limit, DC backupserver)
implements Session {
  Bool mightNeedToMove = False;
  Time timeToMove = Time(0);
  DC origserver = thisDC();

  Unit moveTo(DC dc) { ... } // As before

  Bool order(Int cost) {
    timeToMove = addDuration(now(), limit);
    while (cost > 0) {
      [Cost: 1] cost = cost − 1;
      if (timeValue(now()) > timeValue(timeToMove) && thisDC() != backupserver) {
        this.moveTo(backupserver);
      }
    }
    Bool success = durationValue(deadline()) > 0;
    agent!free(this, success);
    this.moveTo(origserver);
    return success;
  }
}
```

Figure 6.11: Self-monitoring session objects. The session moves to the backupserver if the request runs longer than limit.

using an open workload scenario (in which the clients send periodic requests without synchronizing).

Figure 6.10 shows the Real-Time ABS class SmartAgent which models a load balancing agent which moves sessions to a backup server when the load on the main server increases beyond a certain threshold, namely that the average load of the main server in the last past four time intervals exceeds 50%. This load balancing strategy tries to minimize the amount of work done on the backup server, while maintaining an acceptable quality of service. When the load threshold is reached, the getsession method calls the moveTo method of the session object before the session is returned. When the session is finished, the method free similarly returns the session to the main server.

Figure 6.11 shows the Real-Time ABS class SmartSession which models self-monitoring session objects which move themselves to the backup server if the execution of the current request exceeds a given time limit (which is set at creation time in the example through the constructor parameter limit). Here, the order method initially calculates the threshold execution time timeToMove and moves the session to the backup server once execution time passes the threshold.

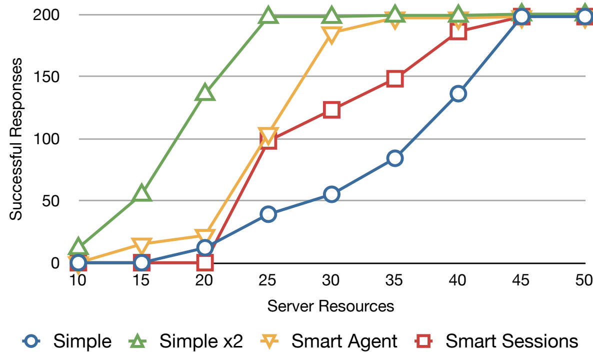


Figure 6.12: Simulation results for the load balancing strategies using a balancing agent (smart agent) and self-monitoring sessions (smart sessions) and for single servers.

***Simulations of Load Balancing Deployment Scenarios*** For the simulations of the server example augmented with load balancing strategies, we added a second deployment component with the same capacity as the primary deployment component. The simulated client job size was chosen so that the capacity of the servers range from complete overloaded to successful completion of all requests.

Figure 6.12 summarizes all three scenarios (single server, load balancing agent, and self-balancing sessions) when the capacity of the deployment components ranges from 10 to 50 resources, as well as a single server with twice the resources (i.e., ranging from 20 to 100). This second single server has the same capacity as the two balanced servers combined and illustrates the efficiency of the different balancing scenarios under the chosen workload.

## 6.6 Example: Load Balancing via Resource Transfer

At midnight on New Year's Eve the behavior of cellphone users briefly changes from normal usage (i.e., a fairly low number of calls and messages) to sending large numbers of SMS messages. We use this phenomenon to motivate and illustrate a complementary approach to load balancing based on the reallocation of (virtualized) resources between deployment components.

The model consists of two cooperating services, TelephoneService and SMSService, and a number of handset clients interacting with these services. The interfaces and implementations of the two services are given in Figure 6.13. The method call will be

```
interface TelephoneServer {
  Unit call(Int calltime);
}

interface SMSServer {
  Unit sendSMS();
}

class TelephoneServer implements TelephoneServer {
  Int callcount = 0;

  Unit call(Int calltime){
    while (calltime > 0) {
      [Cost: 1] calltime = calltime − 1;
      await duration(1, 1);
    }
    callcount = callcount + 1;
  }
}

class SMSServer implements SMSServer {
  Int smscount = 0;

  Unit sendSMS() {
    [Cost: 1] smscount = smscount + 1;
  }
}
```

Figure 6.13: The telephony and SMS services.

invoked *synchronously*; as a parameter the client provides a duration for the call. The method sendSMS will be called *asynchronously*. Note that this model abstracts from many further details which can be added as needed (e.g., a data model, bandwidth, server internals).

The model of the handset clients interoperating with the services is given in Figure 6.14. Client behavior is regulated by a parameter cycle, which determines the frequency of phone calls and messages sent from the handset. Between time $t = 50$ and 70, Handset objects (modeling the behavior of their clients) change to "midnight" behavior and send SMS messages in a rapid pace, otherwise they have "normal" behavior and alternate between sending SMS and making calls.

Simulating this model in a scenario with infinite resources leads to a *purely behavioral model*, where each object acts according to its specification (as in normal Real-Time ABS). Placing the SMS service in an environment with restricted resources leads to observable overload during the midnight window, given a sufficient number of clients to consume all its resources.

In Figure 6.15 the main block defines a scenario where each service runs in its own

```
class Handset (Int cyclelength, TelephoneServer ts, SMSServer smss) {
  Bool call = False;

  Unit normalBehavior() {
    if (timeValue(now()) > 50 && timeValue(now()) < 70) {
      this!midnightWindow();
    }
    else {
      if (call) {
          ts.call(1);
      }
      else {
          smss!sendSMS();
      }
      call = ~ call;
      await duration(cyclelength,cyclelength);
      this!normalBehavior();
    }
  }

  Unit midnightWindow() {
    if (timeValue(now()) >= 70) {
      this!normalBehavior();
    }
    else {
      Int i = 0;
      while (i < 10) {
        smss!sendSMS();
        i = i + 1;
      }
      await duration(1,1);
      this!midnightWindow();
    }
  }

  Unit run(){
    this!normalBehavior();
  }
}
```

Figure 6.14: The Handset class, implementing "New Year's Eve" behavior. Before and after midnight, clients alternate between short calls and sending single messages. During the midnight window ($50 \leq t \leq 70$), ten SMS are sent per cycle.

deployment component with a capacity of 20 resources, and four clients run in the unrestricted root deployment component environment. Dynamic load balancing is implemented by the Balancer class, an instance of which runs in parallel with the service in each component. This class implements a simple load balancing strategy, transferring resources to its partner deployment component when receiving a request message, and monitoring its own load and requesting assistance when needed. More involved or hierarchical schemes for distributing resources among deployment components can

be defined similarly.

Figure 6.16 presents simulation results for this scenario. Results for a scenario without any load balancing is also presented, which shows that the allocated resources are more than sufficient for servicing the normal client behavior, but the SMS service is overloaded during the whole load peak and for another 20 time intervals while catching up with the backlog of delayed messages. In the load balancing scenario, the SMS service is working at capacity during the midnight window but finishes the work backlog two time intervals after the demand spike subsides. After the load on the SMS service returns to normal, the capacity between the two balancers is rebalanced. Note that both scenarios use the identical functional model. The balancing functionality is implemented by two active objects, more elaborate load balancing strategies can be added in similar ways.

## 6.7 Semantics

The operational semantics of Real-Time ABS extended with deployment components and resource consumption is presented as a transition system in an SOS style [10].

### 6.7.1 Runtime Configurations

The runtime syntax is given in Figure 6.17. A *timed configuration* $tcn$ adds a global clock $cl(t)$ to a configuration (where $t$ is a value of type Time). A *configuration* $cn$ is a multiset of objects, invocation messages, futures, and deployment components. The associative and commutative union operator on (timed) configurations is denoted by whitespace and the empty configuration by $\varepsilon$. Note the use of brackets on timed configurations $\{tcn\}$ which will be used when we consider the *whole* configuration and not just some of its terms; i.e., a bracketed configuration will only give a top-level match in the transition system (detailed in Section 6.7.3).

An *object* $obj$ is a term $o(\sigma, p, q)$ where $o$ is the object's identifier, $\sigma$ is a substitution representing the binding of the object's fields, $p$ is an (active) process, and $q$ a *pool of processes*. For substitutions $\sigma$ and process pools $q$, concatenation is denoted by $\sigma_1 \circ \sigma_2$ and $q_1 \circ q_2$, respectively. A *process* $\{\sigma|s\}$ consists of a substitution $\sigma$ of local variable bindings and a list $s$ of statements, or it is *idle*. (We identify any process with an empty statement list with the *idle* process.) We let the fields of an object include *this* and *thisDC*, and the local variables of a process include *deadline* and *destiny* (assuming no name capture). The value of *this* is the identifier of the object and the value of *thisDC* is bound to the object's current deployment component. The value of *deadline* is the remaining duration of the deadline of the process and the value of *destiny* is the address for the return of the process.

```
interface Balancer {
  Unit requestdc(DC comp);
  Unit setPartner(Balancer p);
}

class Balancer implements Balancer {
  Balancer partner = null;

  Unit run() {
    await partner != null;
    while (True) {
      await duration(1, 1);
      Rat ld = thisDC().load(1);
      if (ld > 90) {
        Fut<Unit> r = partner!requestdc(thisDC());
        await r?;
      }
    }
  }

  Unit requestdc(DC comp) {
    DCData total = thisDC().total();
    Rat ld = thisDC().load(1);
    if (ld < 50) {
      thisDC()!transfer(comp, capacity(total) / 3);
    }
  }

  Unit setPartner(Balancer p) {
    partner = p;
  }
}

{
  // Main block
  DC smscomp = new cog DeploymentComponent("smscomp", CPU(50));
  DC telcomp = new cog DeploymentComponent("telcomp", CPU(50));

  [DC: smscomp] SMSServer sms = new cog SMSServer();
  [DC: telcomp] TelephoneServer tel = new cog TelephoneServer();
  [DC: smscomp] Balancer smsb = new cog Balancer();
  [DC: telcomp] Balancer telb = new cog Balancer();
  smsb!setPartner(telb);
  telb!setPartner(smsb);
  new cog Handset(1,tel,sms); new cog Handset(1,tel,sms);
  await duration(1, 1);
  new cog Handset(1,tel,sms); new cog Handset(1,tel,sms);
}
```

Figure 6.15: A resource reallocation strategy and deployment configuration. Without the Balancer objects, the model runs with no functional changes but with a different timing behavior due to overload in the SMS deployment component.

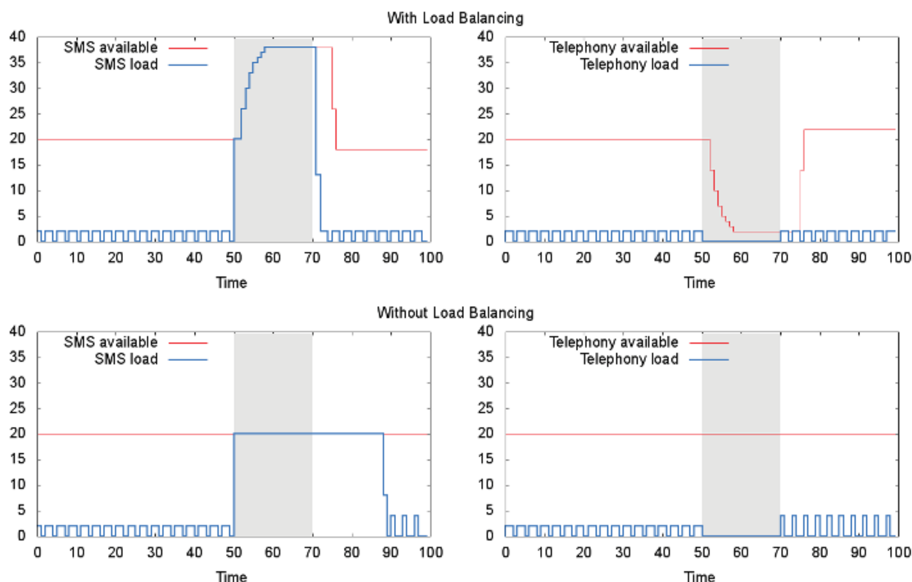


Figure 6.16: Simulation of "New Year's Eve" behavior (SMS load spike between t=50 and t=70), with (top) and without resource balancing (bottom).

In an *invocation message* $m(o, \overline{v}, f, d)$, $m$ is the method name, $o$ the callee, $\overline{v}$ the call's actual parameter values, $f$ the future to which the call's result is returned and $d$ is the provided deadline. A *future* is either an identifier $f$ or a term $f(v)$ with an identifier $f$ and a reply value $v$. For simplicity, classes are not represented explicitly in the semantics, but may be seen as static tables of object layout and method definitions. In a *deployment component* $dc(n, u, k, \overline{h}, \overline{z})$, $dc$ is its identity, $n$ is the total number of available processing resources allocated for the current time interval, $u$ the used resources in the current time interval, k is the number of resources to be allocated for the next time interval, $\overline{h}$ the (possibly empty) sequence of resource usage over time intervals and $\overline{z}$ the (possibly empty) sequence of total allocated resources over time intervals.

The values $v$ are extended with identifiers for the dynamically created objects, futures, and deployment components, statements with **duration2**$(v, v)$ (where the best and worst case expressions must similarly be values instead of expressions), and expressions $e$ with **case2** $v$ $\{\overline{br}\}$ (where the condition must be a value). In the statements $s$, we further assume for simplicity that all method call assignments have deadline annotations as explained in Section 6.2.2 and we let *default*$(T)$ denote a default value of type $T$; e.g., **null** for interface types.

$$
\begin{array}{rcl@{\qquad}rcl}
tcn & ::= & cn\ cl(t) \mid \{cn\ cl(t)\} & v & ::= & o \mid f \mid dc \mid \ldots \\
cn & ::= & \varepsilon \mid obj \mid msg \mid fut \mid cmp \mid cn\ cn & \sigma & ::= & x \mapsto v \mid \sigma \circ \sigma \\
fut & ::= & f \mid f(v) & p & ::= & \{\sigma|s\} \mid idle \\
obj & ::= & o(\sigma, p, q) & q & ::= & \varepsilon \mid p \mid q \circ q \\
msg & ::= & m(o, \overline{v}, f, d) & s & ::= & \textsf{\textbf{duration2}}(v, v) \mid \ldots \\
cmp & ::= & dc(n, u, k, \overline{h}, \overline{z}) & e & ::= & \textsf{\textbf{case2}}\ v\ \{\overline{br}\} \mid \ldots
\end{array}
$$

Figure 6.17: Runtime syntax; here, $o$, $f$, and $dc$ are identifiers for objects, futures, and deployment components, $x$ is the name of a variable, and $d$ is the deadline annotation.

***Initial configuration*** The initial configuration of a program reflects its main block; for a program with main block $\{\overline{T}\ \overline{x}; s\}$ the initial configuration has the form

$$main(a, \{l|s\}, \varepsilon)\ environment(InfCPU, 0, InfCPU, \varepsilon, \varepsilon)\ cl(0)$$

where *main* is an object, *environment* is the default deployment component with unlimited allocated resources in both the current and next time intervals, and $cl(0)$ is the system clock at time 0. In the *main* object, let $a$ be the substitution $\varepsilon[this \mapsto main, thisDC \mapsto environment]$ and $l$ be the substitution $\varepsilon[destiny \mapsto default(\textsf{\textbf{Fut}}\langle\textsf{Unit}\rangle), deadline \mapsto InfDuration], \overline{x} \mapsto default(\overline{T})$. (We assume that for a well-typed program, the main block does not refer to the expressions **this**, **destiny**(), and **deadline**().)

### 6.7.2 The Timed Evaluation of Expressions

Let $\sigma$ be a substitution which binds the name *destiny* to a future identifier, *deadline* to a duration value, *this* to an object identifier, and *thisDC* to the identifier of a deployment component. The evaluation function for expressions $e$ given a substitution $\sigma$ at a time $t$ is defined inductively over the data types of the functional language (see Figure 6.18) and is mostly standard, hence this subsection only contains brief remarks about some of the expressions. For every (user defined) function definition

$$\textsf{\textbf{def}}\ T\ fn(\overline{T\ x}) = e_{fn},$$

the evaluation of a function call $[\![fn(\overline{e})]\!]^t_\sigma$ reduces to the evaluation of the corresponding expression $[\![e_{fn}]\!]^t_{\overline{x} \mapsto \overline{v}}$ when the arguments $\overline{e}$ have already been reduced to ground terms $\overline{v}$. (Note the change in scope. Since functions are defined independently of the context where they are used, we here assume that the expression $e$ does not contain free variables and the substitution $\sigma$ does not apply in the evaluation of $e$.) In the case of pattern matching, variables in the pattern $p$ may be bound to argument values in $v$. Thus the substitution context for evaluating the right hand side $e$ of the branch $p \to e$ extends the current substitution $\sigma$ with bindings that occurred during the pattern matching. Let the function $match(p, v)$ return a substitution such that $match(p, v)(p) = v$ (if there is no match, $match(p, v) = \bot$). For simplicity, we here assume that the evaluation of functional expressions is terminating.

$$
\begin{array}{l}
[\![x]\!]^t_\sigma = \sigma(x)\\
[\![v]\!]^t_\sigma = v\\
[\![\textbf{now}()]\!]^t_\sigma = t\\
[\![Co(\overline{e})]\!]^t_\sigma = Co([\![\overline{e}]\!]^t_\sigma)\\
[\![\textbf{destiny}()]\!]^t_\sigma = \sigma(destiny)\\
[\![\textbf{deadline}()]\!]^t_\sigma = \sigma(deadline)\\
[\![\textbf{this}]\!]^t_\sigma = \sigma(this)\\
[\![\textbf{thisDC}()]\!]^t_\sigma = \sigma(thisDC)\\
[\![fn(\overline{e})]\!]^t_\sigma = \begin{cases} [\![e_{fn}]\!]^t_{\overline{x}\mapsto\overline{v}} & \text{if } \overline{e} = \overline{v}\\ [\![fn([\![\overline{e}]\!]^t_\sigma)]\!]^t_\sigma & \text{otherwise}\end{cases}\\
\\
[\![\textbf{case}\ e\ \{\overline{br}\}]\!]^t_\sigma = [\![\textbf{case2}\ [\![e]\!]^t_\sigma\ \{\overline{br}\}]\!]^t_\sigma\\
[\![\textbf{case2}\ v\ \{p \Rightarrow e;\overline{br}\}]\!]^t_\sigma = \begin{cases} [\![e]\!]^t_{\sigma\circ match(p,br)} & \text{if } match(p,br) \neq \bot\\ [\![\textbf{case2}\ v\ \{\overline{br}\}]\!]^t_\sigma & \text{otherwise}\end{cases}
\end{array}
$$

Figure 6.18: The evaluation of functional expressions.

### 6.7.3 A Transition System for Timed Configurations

Let the transition relation $\rightarrow_t$ capture transitions between timed configurations let $\rightarrow$ represent untimed execution. A timed run is a non-terminating sequence of timed configurations $\{tcn_0\}, \{tcn_1\}, \ldots$ such that $\{tcn_i\} \rightarrow_t \{tcn_{i+1}\}$. Similarly, an untimed run is a possibly terminating sequence of (timed) configurations $tcn_0, tcn_1, \ldots$ such that $tcn_i \rightarrow tcn_{i+1}$. Let $tcn \xrightarrow{!} tcn'$ denote that $tcn'$ is a normal form resulting from a terminating run from the initial configuration $tcn$; i.e., there is no configuration $tcn''$ such that $tcn' \rightarrow tcn''$.

We define a *maximal progress semantics* in which the rules for time advance only apply when untimed execution is blocked; i.e., for a timed configuration $tcn_i$, the relation $\{tcn_i\} \rightarrow_t \{tcn_{i+1}\}$ is defined by $tcn_i \xrightarrow{!} tcn'_i$ and $tcn_{i+1} = \phi(tcn'_i)$, where $\phi$ is one of the auxiliary functions which express the effect of advancing time on the terms of the configuration $tcn'_i$. When auxiliary functions such as $\phi$ are used in the semantics, these are evaluated in between the application of transition rules in a run. Rules apply to subsets of configurations (the standard context rules are not listed). For simplicity we assume that configurations can be reordered to match the left hand side of the rules, i.e., matching is modulo associativity and commutativity as in rewriting logic [13]. Real-Time ABS does not assume that time will always advance; time will never advance in models where execution never runs out of resources and never gets blocked. This corresponds to resource-unaware or infinitely fast models.

***Evaluating guards*** Given a substitution $\sigma$, a time $t$ and a configuration $cn$, we lift the evaluation function for functional expressions to guards and denote by $[\![g]\!]^{t,cn}_\sigma$ an

evaluation function which reduces guards $g$ to data values (here the configuration $cn$ is needed to evaluate future variables). Let $[\![g_1 \wedge g_2]\!]_\sigma^{t,cn} = [\![g_1]\!]_\sigma^{t,cn} \wedge [\![g_2]\!]_\sigma^{t,cn}$, $[\![x?]\!]_\sigma^{t,cn} =$ $\mathsf{True}$ if $[\![x]\!]_\sigma^{t,cn} = f$ and $f(v) \in cn$ for some value $v$ (i.e., the future already has a value), otherwise $f \in cn$ and we let $[\![x?]\!]_\sigma^{t,cn} = \mathsf{False}$. Guards that are Boolean expressions reduce as expected: $[\![e]\!]_\sigma^{t,cn} = [\![e]\!]_\sigma^{t}$ (note that such guards can change their value only if they refer to the object state).

***Auxiliary functions*** If the class of an object $o$ has a method $m$, we let $bind(m, o, \overline{v}, f, d)$ return a process resulting from the activation of $m$ on $o$ with actual parameters $\overline{v}$, an associated future $f$, and a deadline $d$. If the binding succeeds, the local variable *destiny* in the new process is bound to $f$, *deadline* is bound to $d$, and the method's formal parameters are bound to $\overline{v}$. The function $select(q, \sigma, cn)$ schedules a process which is ready to execute from the process queue $q$ of an object $o(\sigma, idle, q)$ in a configuration $cn$. The function $atts(C, \overline{v}, o, dc)$ returns the initial substitution $\sigma$ for the fields of a new instance $o$ of class $C$, in which the formal parameters are bound to $\overline{v}$, the field *this* is bound to the object identity $o$ and the field *thisDC* to the deployment component $dc$. The function $init(C)$ returns an activation (process) of the *init* method of $C$, if defined. Otherwise it returns the *idle* process. The predicate $fresh(n)$ asserts that a name $n$ is globally unique (where $n$ may be an identifier for an object, a future, or a deployment component). The definition of these functions is straightforward but requires that the class table is explicit in the semantics, which we have omitted for simplicity.

***Transition rules*** Transition rules transform configurations into new configurations, and are given in Figures 6.19 and 6.20. In the semantics, different assignment rules are defined for side effect free expressions (Assign1 and Assign2), object creation (New-Object1 and New-Object2), method calls (Async-call), and future dereferencing (Read-Fut). We conventionally write $a$ to denote the substitution which maps fields to values in an object and $l$ to denote the substitution which maps local variables to values in a process. Annotations are used to provide a deadline, a cost, and to associate objects with deployment components. (In the implementation, these annotations are generated with default values by the compiler if they are not explicitly given in the source code.)

Rule Skip consumes a **skip** in the active process. Here and in the sequel, the variable $s$ will match any (possibly empty) statement list. Rules Assign1 and Assign2 assign the value of expression $e$ to a variable $x$ in the local variables $l$ or in the fields $a$, respectively. Rules Cond1 and Cond2 cover the two cases of conditional statements. (We omit the standard rule which unfolds while-loops into the conditional.)

Note that in the Activate rule, in order to evaluate guards on futures, the entire configuration $cn$ is passed to the *select* function. This explains the use of brackets

(Skip)

$$o(a, \{l \mid \textbf{skip}; s\}, q) \rightarrow o(a, \{l \mid s\}, q)$$

(Activate)

$$\frac{p = select(q, a, cn)}{\{o(a, idle, q) \;\; cn \;\; cl(t)\} \rightarrow \{o(a, p, (q \setminus p)) \;\; cn \;\; cl(t)\}}$$

(Suspend)

$$o(a, \{l \mid \textbf{suspend}; s\}, q) \rightarrow o(a, idle, \{l \mid s\} \circ q)$$

(Assign1)

$$\frac{x \in dom(l)}{o(a, \{l \mid x = e; s\}, q) \;\; cl(t) \rightarrow o(a, \{l[x \mapsto [\![e]\!]^t_{a \circ l}] \mid s\}, q) \;\; cl(t)}$$

(Assign2)

$$\frac{x \in dom(a)}{o(a, \{l \mid x = e; s\}, q) \;\; cl(t) \rightarrow o(a[x \mapsto [\![e]\!]^t_{a \circ l}], \{l \mid s\}, q) \;\; cl(t)}$$

(Cond1)

$$\frac{[\![e]\!]^t_{a \circ l}}{o(a, \{l \mid \textbf{if}\ e\ \{s_1\}\ \textbf{else}\ \{s_2\}; s\}, q) \;\; cl(t) \rightarrow o(a, \{l \mid s_1; s\}, q) \;\; cl(t)}$$

(Cond2)

$$\frac{\neg [\![e]\!]^t_{a \circ l}}{o(a, \{l \mid \textbf{if}\ e\ \{s_1\}\ \textbf{else}\ \{s_2\}; s\}, q) \;\; cl(t) \rightarrow o(a, \{l \mid s_2; s\}, q) \;\; cl(t)}$$

(Await1)

$$\frac{[\![e]\!]^{t,cn}_{a \circ l}}{\{o(a, \{l \mid \textbf{await}\ e; s\}, q) \;\; cl(t) \;\; cn\} \rightarrow \{o(a, \{l \mid s\}, q) \;\; cl(t) \;\; cn\}}$$

(Await2)

$$\frac{\neg [\![e]\!]^{t,cn}_{a \circ l}}{\{o(a, \{l \mid \textbf{await}\ e; s\}, q) \;\; cl(t) \;\; cn\} \rightarrow \{o(a, \{l \mid \textbf{suspend}; \textbf{await}\ e; s\}, q) \;\; cl(t) \;\; cn\}}$$

(Async-Call)

$$\frac{an = \texttt{Deadline:}\ e', an' \quad [\![e]\!]^t_{a \circ l} = o' \quad [\![e']\!]^t_{a \circ l} = d \quad fresh(f)}{o(a, \{l \mid [an]\ x = e!m(\bar{e}); s\}, q) \;\; cl(t) \rightarrow o(a, \{l \mid [an']\ x = f; s\}, q) \;\; m(o', [\![\bar{e}]\!]^t_{a \circ l}, f, d) \;\; f \;\; cl(t)}$$

(Bind-Mtd)

$$\frac{q' = bind(m, o, \bar{v}, f, d) \circ q}{o(a, \{l \mid s\}, q) \;\; m(o, \bar{v}, f, d) \rightarrow o(a, \{l \mid s\}, q')}$$

(Return)

$$\frac{f = l(destiny)}{o(a, \{l \mid \textbf{return}(e); s\}, q) \;\; f \;\; cl(t) \rightarrow o(a, idle, q) \;\; f([\![e]\!]^t_{a \circ l}) \;\; cl(t)}$$

(Read-Fut)

$$\frac{f = [\![e]\!]^t_{a \circ l}}{o(a, \{l \mid x = e.\textbf{get}; s\}, q) \;\; f(v) \;\; cl(t) \rightarrow o(a, \{l \mid x = v; s\}, q) \;\; f(v) \;\; cl(t)}$$

(Duration1)

$$\frac{v_1 = [\![e_1]\!]^t_{a \circ l} \quad v_2 = [\![e_2]\!]^t_{a \circ l}}{o(a, \{l \mid \textbf{duration}(e_1, e_2); s\}, q) \;\; cl(t) \rightarrow o(a, \{l \mid \textbf{duration2}(v_1, v_2); s\}, q) \;\; cl(t)}$$

(Duration2)

$$\frac{v_1 \leq 0}{o(a, \{l \mid \textbf{duration2}(v_1, v_2); s\}, q) \rightarrow o(a, \{l \mid s\}, q)}$$

Figure 6.19: Semantics for Real-Time ABS with deployment components and resource consumption (1).

(New-Object1)
$$\frac{fresh(o') \quad an = \texttt{DC:}\ e', an' \quad [\![e']\!]^t_{a\circ l} = dc \quad p' = init(C) \quad a' = atts(C, [\![\overline{e}]\!]^t_{a\circ l}, o', dc)}{o(a, \{l \mid [an]\ x = \textbf{new}\ C(\overline{e}); s\}, q)\ cl(t) \to o(a, \{l \mid [an']\ x = o'; s\}, q)\ o'(a', p', \emptyset)\ cl(t)}$$

(New-Object2)
$$\frac{fresh(o') \quad a(thisDC) = dc \quad p' = init(C) \quad a' = atts(C, [\![\overline{e}]\!]^t_{a\circ l}, o', dc)}{o(a, \{l \mid x = \textbf{new}\ C(\overline{e}); s\}, q)\ cl(t) \to o(a, \{l \mid x = o'; s\}, q)\ o'(a', p', \emptyset)\ cl(t)}$$

(New-DC)
$$\frac{fresh(dc) \quad [\![e]\!]^t_{a\circ l} = n}{o(a, \{l \mid x = \textbf{new}\ \texttt{DeploymentComponent}(e_0, e); s\}, q)\ cl(t) \to o(a, \{l \mid x = dc; s\}, q)\ dc(n, 0, n, \varepsilon, \varepsilon)\ cl(t)}$$

(Emp-Annotation)
$$o(a, \{l \mid [\,\varepsilon\,]\ s\}, q) \to o(a, \{l \mid s\}, q)$$

(Cost1)
$$\frac{a(thisDC) = dc \quad an = \texttt{Cost:}\ e, an' \quad [\![e]\!]^t_{a\circ l} = c \quad c \leq n - u \quad o(a, \{l \mid [an']\ s\}, q)\ cl(t)\ cn \to o(a', p', q')\ cl(t)\ cn'}{o(a, \{l \mid [an]\ s\}, q)\ dc(n, u, k, \overline{h}, \overline{z})\ cl(t)\ cn \to o(a', p', q')\ dc(n, u + c, k, \overline{h}, \overline{z})\ cl(t)\ cn'}$$

(Cost2)
$$\frac{a(thisDC) = dc \quad an = \texttt{Cost:}\ e', an' \quad [\![e']\!]^t_{a\circ l} = c \quad c > n - u \quad n \neq u \quad c' = c - (n - u) \quad an'' = \texttt{Cost:}\ c', an'}{o(a, \{l \mid [an]\ s\}, q)\ dc(n, u, k, \overline{h}, \overline{z})\ cl(t)\ cn \to o(a, \{l \mid [an'']\ s\}, q)\ dc(n, n, k, \overline{h}, \overline{z})\ cl(t)\ cn}$$

(Total)
$$\frac{fresh(f) \quad an = \texttt{Deadline:}\ e', an' \quad [\![e]\!]^t_{a\circ l} = dc}{o(a, \{l \mid [an]\ x = e!\textbf{total}(); s\}, q)\ dc(n, u, k, \overline{h}, \overline{z})\ cl(t) \to o(a, \{l \mid [an']\ x = f; s\}, q)\ dc(n, u, k, \overline{h}, \overline{z})\ f(n)\ cl(t)}$$

(Move)
$$\frac{[\![e]\!]^t_{a\circ l} = dc}{o(a, \{l \mid \textbf{moveto}(e); s\}, q)\ cl(t) \to o(a[thisDC \mapsto dc], \{l \mid s\}, q)\ cl(t)}$$

(Transfer)
$$\frac{fresh(f) \quad an = \texttt{Deadline:}\ e', an' \quad [\![e]\!]^t_{a\circ l} = dc \quad [\![e']\!]^t_{a\circ l} = dc' \quad [\![e'']\!]^t_{a\circ l} = i \quad i' = min(i, k)}{o(a, \{l \mid [an]\ x = e!\textbf{transfer}(e', e''); s\}, q)\ dc(n, u, k, \overline{h}, \overline{z})\ dc'(n', u', k', \overline{h'}, \overline{z'})\ cl(t) \to o(a, \{l \mid [an']\ x = f; s\}, q)\ dc(n, u, k - i', \overline{h}, \overline{z})\ dc'(n', u', k' + i', \overline{h'}, \overline{z'})\ f(i')\ cl(t)}$$

(Load)
$$\frac{an = \texttt{Deadline:}\ e', an' \quad [\![e]\!]^t_{a\circ l} = dc \quad [\![e']\!]^t_{a\circ l} = i \quad fresh(f) \quad v = avg(\overline{h}, \overline{z}, i)}{o(a, \{l \mid [an]\ x = e!\textbf{load}(e'); s\}, q)\ dc(n, u, k, \overline{h}, \overline{z})\ cl(t) \to o(a, \{l \mid [an']\ x = f; s\}, q)\ dc(n, u, k, \overline{h}, \overline{z})\ f(v)\ cl(t)}$$

(Run-Inside-Interval)
$$\frac{cn\ cl(t) \xrightarrow{!} cn'\ cl(t) \quad 0 < d \leq mte(cn', t) \quad \lfloor t \rfloor = \lfloor t + d \rfloor}{\{cn\ cl(t)\} \to_t \{timeAdv(cn', d)\ cl(t + d)\}}$$

(Run-To-New-Interval)
$$\frac{cn\ cl(t) \xrightarrow{!} cn'\ cl(t) \quad 0 < d \leq mte(cn', t) \quad \lceil t \rceil = t + d}{\{cn\ cl(t)\} \to_t \{timeAdv(rscRefill(cn'), d)\ cl(t + d)\}}$$

Figure 6.20: Semantics for Real-Time ABS with deployment components and resource consumption (2).

in this rule, which ensures that $cn$ is bound to the full configuration and not just a part of the configuration. The same approach is used to evaluate guards in the rules Await1 and Await2 below.

Rule Suspend enables cooperative scheduling and suspends the active process to the process pool, leaving the active process *idle*. Rule Await1 consumes the **await** $g$ statement if $g$ evaluates to true in the current state of the object, rule Await2 adds a **suspend** statement to the process if the guard evaluates to false.

In rule Bind-Mtd the function $bind(m, o, \overline{v}, f, d)$ binds a method call in the class of the callee $o$. This results in a new process $\{l \mid s\}$ which is placed in the queue, where $l(destiny) = f$, $l(deadline) = d$, and where the formal parameters of $m$ are bound to $\overline{v}$ in $l$.

*Method calls.* Rule Async-Call sends an invocation message to $[\![e]\!]_{a \circ l}$ with the unique identity $f$ of a new future (since $fresh(f)$), the method name $m$, actual parameters $\overline{v}$, and deadline $d$. The identifier of the new future is placed in the configuration, and is bound to a return value in Return. Rule Return places the evaluated return expression in the future associated with the executing process, and stops the execution. Rule Read-Fut dereferences a future on the form $f(v)$. Note that if the future lacks a return value, it is of the form $f$ and the reduction in this object is *blocked*.

*Durations.* In rule Duration1, the statement **duration**$(e_1, e_2)$ is transformed to **duration2**$(v_1, v_2)$ by reducing the expressions $e_1$ and $e_2$ to their values. In rule Duration2, this statement *blocks execution on the object* until the best case execution time $v_1$ has passed. This depends on the *time advance function*; the effect of advancing the time by a duration $d$ is that **duration2**$(v_1, v_2)$ is reduced to **duration2**$(v_1 - d, v_2 - d)$. The *maximal time elapse function* similarly ensures that time cannot pass beyond duration $v_2$ before the statement has been executed. These two functions, which control time advance in the semantics, are discussed in detail below.

*Object creation.* Rules New-Object1 and New-Object2 create a new object with a unique identifier $o'$. The object's fields are given default values by $atts(C, [\![\overline{e}]\!]^t_{a \circ l}, o', dc)$, extended with the actual values $[\![\overline{e}]\!]^t_{a \circ l}$ for the class parameters (evaluated in the context of the creating process), $o'$ for *this* and $dc$ for *thisDC*. In order to instantiate the remaining attributes, the process $init(C)$ will be active (this function returns *idle* if the *init* method is unspecified in the class $C$, and it asynchronously calls `run` if the latter is specified). New-Object1 deals with deployment component annotations.

*Deployment components.* Rule New-DC creates a new deployment component $dc(n, 0, n, \varepsilon, \varepsilon)$ where $dc$ is a unique identifier, $n$ the capacity of the deployment component, and $\varepsilon$ an empty list. Rule Emp-Annotation removes an empty list of annotations. The rules Cost1 and Cost2 capture the reduction of an object $o$ in which the head of the statement list in the active process has a cost annotation with expression $e$. Rule Cost1 covers the case in which the deployment component has enough resources to execute the statement inside the time interval. Rule Cost2 covers the case in which

the deployment component does not have enough resources to execute the statement inside the time interval; i.e., the required resources $c$ are larger than the available resources $n - u$. Since we work with processing resources (as opposed to, e.g., memory resources which must be obtained atomically), we allow execution to take several time intervals, and let the cost expression be gradually reduced. In both rules, the consumed resources are added to $u$ in $dc$. Rule Total assigns to the variable $x$ the total amount of resources in $dc$ in the current time interval. Observe that the deadline annotation is ignored, since the result is obtained in the same execution step. This also applies for **load** and **transfer**. Rule Move changes the deployment component associated with the object $o$ to $dc$. The rule Transfer reallocates $i'$ resources from $dc$ to $dc'$ to be effective in the next time interval. Note that if the deployment component does not have $i$ resources available for the next time interval, only the available amount $k$ will be reallocated. The rule Load calculates and assigns to the variable $x$ the average percent of used resources in $dc$ during the last $i$ time intervals. Let $nth(\overline{h}, n)$ select the $n$'th element of a sequence $\overline{h}$, and $length(\overline{h})$ the number of elements in $\overline{h}$. It may be the case that $length(\overline{h}) < i$, in which case we can only calculate **load**$(length(\overline{h}))$. Therefore, we define the average resource load in percentage (scaled from 0 to 100) as follows:

$$avg(\overline{h}, \overline{z}, i) = \sum_{j=1}^{min(i, length(\overline{h}))} \frac{nth(\overline{h}, j)}{nth(\overline{z}, j)} \times \frac{100}{min(i, length(\overline{h}))}.$$

***Time advance*** Time advance in the system is specified by the two rules Run-Inside-Interval and Run-To-New-Interval. Our model of time is based on maximal progress, so time will only advance when execution is otherwise blocked. The rule Run-Inside-Interval captures time advance which does not influence the resource availability in the deployment components of the system, and the rule Run-To-New-Interval captures the case when the resources in the deployment components should be "refilled" for the next time interval.

Following the approach of Real-Time Maude [21], we define an auxiliary function $mte(cn, t)$ which computes the *maximum time elapse* of a configuration $cn$ at time $t$, and an auxiliary function $timeAdv(cn, d)$ which captures the effect on a configuration $cn$ of advancing time by a duration $d$. For any configuration $cn$ and time $t$, the rules Run-Inside-Interval and Run-To-New-Interval allow time to advance by a duration $d \leq mte(cn, t)$. However, we are not interested in advancing time by a duration 0, which would leave the system in the same configuration. The definition of $mte$ ensures that the time for renewing resources in the deployment components is never bypassed. When time advances to the next time interval, the auxiliary function $rscRefill(cn)$ is used to capture the effect of time advance on the deployment components in $cn$.

The auxiliary functions $mte$, $timeAdv$, and $rscRefill$ are defined in Figure 6.21. These functions are recursively defined by cases over the system configuration; the

interesting cases are objects and deployment components since these exhibit time-dependent behavior. Additional subscripted functions which apply to elements of the objects are similarly defined by cases for processes, statement, and guards.

The function *mte* calculates the largest amount by which time can advance such that no "interesting" occurrence will be missed in any object or deployment component (e.g., a worst-case duration expires or the deployment components need to be refilled). To ensure maximal progress, the maximum time elapse is 0 for enabled statements which are not time-dependent and infinite if the statement is not enabled, since time may pass when the object is blocked. A statement is not enabled if it has a cost annotation or is otherwise blocked. Thus, for a process which has a cost annotation for its head statement, time must advance before the process can proceed; the maximum time elapse of this process is infinite. Hence, *mte* returns the minimum time increment that makes some object become "unstuck", either by letting its active process continue or enabling one of its suspended processes.

The function *timeAdv* updates the active and suspended processes of all objects, decrementing the values of all deadline variables and **duration2** statements at the head of the statement list in processes. The function *rscRefill* captures the effect of time advance on the deployment components; the available resources $n$ are refilled according to the amount of resources in $k$, and the histories of resource consumption $\overline{h}$ and of total allocated resources $\overline{z}$ are extended with the used resources $u$ and the current total resources $n$ of the previous time interval, respectively.

## 6.8 Related and Future Work

The concurrency model of ABS combines concurrent objects from Creol [5,6] with concurrent object groups [22] and is reminiscent of Actors [7] and Erlang [9] processes: Object groups are inherently concurrent, conceptually each group has a dedicated processor, and there is at most one activity in a group at any time. This concurrency model has attracted attention as an alternative to multi-thread concurrency in object-orientation (e.g., [4]), and been integrated with, e.g., Java [23] and Scala [8]. Concurrent objects support compositional verification of concurrent software [6, 24], in contrast to multi-threaded object systems. Their inherent compositionality allows concurrent objects to be naturally distributed on different locations, because only an object's local state is needed to execute its methods. A particular feature of ABS, inherited from Creol, is its cooperative scheduling of method activations inside the object groups. In order to capture the timing of object-oriented models, Real-Time ABS [17] extends ABS and its tool suite to combine real-time with concurrent object models.

In the authors' early work on deployment components [14, 15], the execution cost was fixed in the language semantics; following an idea proposed in [16], resource con-

$$
\begin{array}{l}
mte(cn_1\ cn_2, t) = min(mte(cn_1, t), mte(cn_2, t)) \\
mte(o(a,p,q),t) = \begin{cases} mte_p(p,t) & \text{if } p \neq idle \\ mte_p(q,t) & \text{if } p = idle \end{cases} \\
mte(dc(n,u,k,\overline{h},\overline{z}),t) = \lfloor t+1 \rfloor - t \\
mte(cn,t) = \infty \qquad\qquad \text{otherwise}
\end{array}
$$

$$
\begin{array}{l}
mte_p(q_1 \circ q_2, t) = min(mte(q_1, t), mte(q_2, t)) \\
mte_p(\{l|s\},t) = \begin{cases} w & \text{if } s = \textbf{duration2}(b,w); s_2 \\ mte_g(g,t) & \text{if } s = \textbf{await } g; s_2 \\ 0 & \text{if } s \text{ is enabled} \\ \infty & \text{otherwise} \end{cases} \\
mte_p(q) = \infty \qquad\qquad \text{otherwise}
\end{array}
$$

$$
mte_g(g,t) = \begin{cases} max(mte_g(g_1,t), mte_g(g_2,t)) & \text{if } g = g_1 \wedge g_2 \\ 0 & \text{if } g \text{ evaluates to true} \\ \infty & \text{otherwise} \end{cases}
$$

$$
\begin{array}{l}
timeAdv(cn_1\ cn_2, d) = timeAdv(cn_1, d)\ timeAdv(cn_2, d) \\
timeAdv(o(a,p,q),d) = o(a, timeAdv_p(p,d), timeAdv_p(q,d)) \\
timeAdv(cn,d) = cn \qquad\qquad \text{otherwise}
\end{array}
$$

$$
\begin{array}{l}
timeAdv_p((q_1 \circ q_2), d) = timeAdv_p(q_1, d), timeAdv_p(q_2, d) \\
timeAdv_p(\{l|s\}, d) = \{l[deadline \mapsto l(deadline) - d]|timeAdv_s(s,d)\} \\
timeAdv_p(q,d) = q \qquad\qquad \text{otherwise}
\end{array}
$$

$$
timeAdv_s(s,d) = \begin{cases} \textbf{duration2}(b-d, w-d) & \text{if } s = \textbf{duration2}(b,w) \\ \textbf{await } timeAdv_g(g,d) & \text{if } s = \textbf{await } g \\ timeAdv_s(s_1,d); s_2 & \text{if } s = s_1; s_2 \\ s & \text{otherwise} \end{cases}
$$

$$
timeAdv_g(g,d) = \begin{cases} timeAdv_g(g_1,d) \wedge timeAdv_g(g_2,d) & \text{if } g = g_1 \wedge g_2 \\ \textbf{duration2}(b-d, w-d) & \text{if } \ g = \textbf{duration2}(b,d) \\ g & \text{otherwise} \end{cases}
$$

$$
\begin{array}{l}
rscRefill(cn_1\ cn_2) = rscRefill(cn_1)\ rscRefill(cn_2) \\
rscRefill(dc(n,u,k,\overline{h},\overline{z})) = dc(k,0,k,u \circ \overline{h}, n \circ \overline{z}) \\
rscRefill(cn) = cn \qquad\qquad \text{otherwise}
\end{array}
$$

Figure 6.21: Functions controlling the advancement of time and its effect on the system configuration.

sumption is expressed in our paper in terms of optional annotations with user-defined expressions which relate to the local state and the input parameters to methods. This way, the cost of execution in the model may be adapted by the modeler to a specific cost scenario. This allows us to abstractly model the effect of deploying concurrent objects on deployment components with different amounts of allocated resources at an early stage in the software development process, before modeling the detailed control flow of the targeted system.

Whereas this paper has focused on processing resources, initial complementary work addresses deployment components with restricted memory [25] and bandwidth [26]. A more abstract approach to user-defined resource management is discussed in [27], in which the user also specifies when resources are *released* during the execution. Modeling other resource types, as well as the semantics of more than one resource type in a model, is an area of ongoing research for the authors in the scope of the EU FP7 project Envisage, which will extend the approach taken in this paper to cloud computing, service-level agreements, code generation, and monitoring [28]. Preliminary work suggests that annotations can be used in a similar way for other resources, and that the semantics and existing interpreter can be augmented with a generic framework for handling resources.

There is an extensive literature on formal models of locations and mobility based on, e.g., agents, ambient calculi, and process algebras. These models are typically concerned with maintaining correct interactions with respect to, e.g., security, link failure, or location failure. Among non-functional properties, access to shared resources have been studied through type and effect systems (e.g., [29, 30]), QoS-aware processes proposed for negotiating contracts [31], and type-based space control for space-aware processes [32]. Closer to our work, timed synchronous CCS-style processes can be compared for speed using faster-than bisimulation [33], albeit without notions of mobility or location. We are not aware of other formal models connecting reallocatable (virtual) processing capacities to locations.

Techniques for prediction or analysis of non-functional properties are based on either *measurement*, *simulation*, or *modeling* [34]. Measurement-based approaches can only be applied when an implementation already exists (i.e., fairly late in the software life-cycle), using dedicated profiling or tracing tools like JMeter or LoadRunner. Whereas simulations are traditionally done in programming languages (e.g., SIMULA), domain-specific simulation packages and dedicated simulators are very efficient inside their specific application domain but are less flexible [34]. Related work on simulation tools for virtualized resources in cloud computing are typically reminiscent of network simulators. A number of testing techniques and tools for cloud-based software systems are surveyed in [35]. In particular, CloudSim [36] and ICanCloud [37] are simulation tools using virtual machines to simulate cloud environments. CloudSim is a fairly mature tool which has already been used for a number of papers, but it is restricted to simulations on a single computer. In contrast, ICanCloud supports dis-

tribution on a cluster. Additionally CloudSim was originally based on GridSim [38], a toolkit for modeling and simulations of heterogeneous Grid resources. EMUSIM [39] is an integrated tool that uses AEF [40] (Automated Emulation Framework) to estimate performance and costs for an application by means of emulations to produce improved input parameters for simulations in CloudSim. Compared to these approaches, our work is based on a formal semantics and aims to support the developer of software applications at an early phase in the development process. The approach of this paper has been encouragingly compared to specialized simulation tools and to measurements on deployed code in two larger case studies addressing resource management in the cloud; an ABS model of the Montage case study [41] is presented in [42] and compared to results from specialized simulation tools and a large ABS model of the Fredhopper Replication Server has been compared to measurements on the deployed system in [43].

Model-based approaches allow abstraction from specific system intricacies, but depend on parameters provided by domain experts [44]. A survey of model-based performance analysis techniques is given in [45]. Formal systems using process algebra, Petri Nets, game theory, and timed automata have been used in the embedded software domain (e.g., [46–48]), but also to the schedulability of processes in concurrent objects [49, 50]. The latter work complements ours as it does not consider restrictions on shared deployment resources, but studies the schedulability of method activations in the context of concurrent objects as found in Real-Time ABS. User-defined schedulers for concurrent objects were introduced for Real-Time ABS in [17], using optional scheduling annotations and defaults.

Performance evaluation for component-based systems is surveyed in [51]. UML has been extended with a profile for schedulability, performance, and time (SPT) and combined with a methodology for software performance engineering (SPE) [52]. Using the UML SPT profile, Petriu and Woodside [53] informally define the Core Scenario Model (CSM) to solve questions that arise in performance model building. CSM has a notion of resource context, which reflects an operation's set of resources. CSM aims to bridge the gap between UML and techniques to generate performance models [45]. Closer to our work is the extension of VDM++ for embedded real-time systems by M. Verhoef [54] and the Palladio component model by R. Reussner *et al.* [55, 56]. In Verhoef's work, static architectures are explicitly modeled using CPUs and buses. The approach uses fixed resources targeting the embedded domain, namely processor cycles bound to the CPUs. In Palladio, components with explicit resource requirements are deployed on a static architecture where nodes have resources such as CPU, memory, and cache. In contrast to our work, Palladio uses probabilistic finite state machines and abstract from branch conditions and loop iterations. Components in Palladio are assumed to be stateless, but recent work considers an extension to stateful components [57]. Both approaches support simulation-based analysis, which is stochastic for Palladio. Both approaches consider several resources, in contrast

to our work which here focuses on CPU. However, these approaches are restricted to static deployment scenarios. Our work goes beyond static scenarios to consider dynamic deployment and load balancing.

Other interesting lines of research relating to our work are techniques for static cost analysis (e.g., [58, 59]) and symbolic execution [60] for object-oriented programs, and statistical model checking [61, 62]. Since Real-Time ABS is fully formalized, it is interesting to see how such formal analysis techniques can be applied to obtain stronger analysis results than simulations. However, most tools for cost analysis and symbolic execution only consider sequential and untimed programs. In addition, programs must be fully developed before automated cost analysis can be applied. COSTABS [20] is a cost analysis tool for ABS which supports *concurrent* object-oriented programs, based on a novel notion of cost center. Our approach, in which the modeler specifies resource consumption in terms of cost annotations, could be supported by COSTABS to automatically derive cost annotations for the parts of a model that are fully implemented (see Example 16). In collaboration with Albert *et al.*, this approach has been applied for memory analysis of ABS models [25]. However, the generalization of that work for processing resources as well as for general, user-defined cost models, and its integration into the software development process currently remains future work. The separation of concerns between the resource capacity of the deployment layer and the resource consumption of the imperative layer may allow cost analysis and symbolic execution of concurrent timed programs. Extending our tool with symbolic execution allow the approximation of best- and worst-case response times for different deployment scenarios, depending on the available resources and the user load.

The work presented in this paper is based on a *maximal progress* semantics. A case study of the Fredhopper Replication Server [43], where costs were obtained by averaging observations from a real system, suggests that this gives fairly realistic results. Our framework has been extended to Monte Carlo simulations by adding a seed to the simulation tool [16]. An interesting extension of our work is to support statistical model checking [61, 62], for example by combining PVeStA [63] with our simulation tool in Maude. However, a stochastic model requires that meaningful probabilities are assigned to the different transitions of the language interpreter. One approach could be to assign probabilistic information to each deployment component, refining the notion of maximal progress. In addition, it is interesting to use stochastic modeling to specify end-user scenarios for our models.

## 6.9 Conclusion

This paper presents a simple and flexible approach to integrating deployment architectures and resource consumption into executable object-oriented models. The approach is based on a separation of concerns between the resource cost of performing computa-

tions and the resource capacity of the deployment architecture. The paper considers resources which abstractly reflect execution: each deployment component has a resource capacity per time interval and each computation step has a cost, specified by a user-defined cost expression or by a default. This separation of concerns between cost and capacity allows the performance of a model to be easily compared for a range of deployment choices. By comparing deployment scenarios, many interesting questions concerning performance can be addressed already at an early phase of the software design.

The integration of deployment architectures into software models further allows application-level resource management policies to become an integral part of the software design. For deployment scenarios reflecting fixed architectures, it is natural to define the architecture as part of a model's main block. For deployment scenarios reflecting dynamic architectures, new deployment components may be dynamically created to model, e.g., virtualized machines initialized through a middleware layer or on the cloud. This paper explores two complementary approaches to load balancing between existing deployment components as part of the application-level resource management, both based on allowing objects to inspect the load of different parts of the deployment architecture. First, concurrent object groups may move between deployment components and, second, resources may be reallocated between deployment components.

Technically, the paper presents an extension of Real-Time ABS with a deployment layer, including linguistic primitives to express dynamic deployment architectures and resource management at the abstraction level of the modeling language as well as optional annotations with user-defined cost expressions to capture resource consumption. These primitives have been fully integrated with Real-Time ABS, which combines real-time and object-oriented models. The paper presents a complete formal semantics for the extended language and a number of examples to illustrate its usage. The presented semantics has been used to extend the ABS tool suite, which has been applied to obtain simulation results concerning performance for the presented examples.

Whereas most work on performance either specify timing or cost as part of the model (assuming a fixed deployment architecture) or measure the behavior of the compiled code deployed on an actual deployment architecture, the approach presented in this paper addresses a need in formal methods to capture models which vary over the underlying deployment architectures, for example to model deployment variability in software product lines and resource management of virtualized resource management for the cloud.

# Bibliography


[1] K. Pohl, G. Böckle, F. Van Der Linden, Software Product Line Engineering: Foundations, Principles, and Techniques, Springer, 2005.

[2] S. M. Yacoub, Performance analysis of component-based applications, in: G. J. Chastek (Ed.), Proc. Second International Conference on Software Product Lines (SPLC'02), Vol. 2379 of Lecture Notes in Computer Science, Springer, 2002, pp. 299–315.

[3] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, M. Steffen, ABS: A core language for abstract behavioral specification, in: B. Aichernig, F. S. de Boer, M. M. Bonsangue (Eds.), Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010), Vol. 6957 of Lecture Notes in Computer Science, Springer, 2011, pp. 142–164.

[4] D. Caromel, L. Henrio, A Theory of Distributed Object, Springer, 2005.

[5] E. B. Johnsen, O. Owe, An asynchronous communication model for distributed concurrent objects, Software and Systems Modeling 6 (1) (2007) 35–58.

[6] F. S. de Boer, D. Clarke, E. B. Johnsen, A complete guide to the future, in: R. de Nicola (Ed.), Proc. 16th European Symposium on Programming (ESOP'07), Vol. 4421 of Lecture Notes in Computer Science, Springer, 2007, pp. 316–330.

[7] G. A. Agha, ACTORS: A Model of Concurrent Computations in Distributed Systems, The MIT Press, Cambridge, Mass., 1986.

[8] P. Haller, M. Odersky, Scala actors: Unifying thread-based and event-based programming, Theoretical Computer Science 410 (2–3) (2009) 202–220.

[9] J. Armstrong, Programming Erlang: Software for a Concurrent World, Pragmatic Bookshelf, 2007.

[10] G. D. Plotkin, A structural approach to operational semantics, Journal of Logic and Algebraic Programming 60-61 (2004) 17–139.

[11] D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, P. Y. H. Wong, Modeling spatial and temporal variability with the HATS abstract behavioral modeling language, in: M. Bernardo, V. Issarny (Eds.), Proc. 11th Intl. School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM 2011), Vol. 6659 of Lecture Notes in Computer Science, Springer, 2011, pp. 417–457.

[12] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. L. Talcott (Eds.), All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, Vol. 4350 of Lecture Notes in Computer Science, Springer, 2007.

[13] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, Theoretical Computer Science 96 (1992) 73–155.

[14] E. B. Johnsen, O. Owe, R. Schlatte, S. L. Tapia Tarifa, Validating timed models of deployment components with parametric concurrency, in: B. Beckert, C. Marché (Eds.), Proc. International Conference on Formal Verification of Object-Oriented Software (FoVeOOS'10), Vol. 6528 of Lecture Notes in Computer Science, Springer, 2011, pp. 46–60.

[15] E. B. Johnsen, O. Owe, R. Schlatte, S. L. Tapia Tarifa, Dynamic resource reallocation between deployment components, in: J. S. Dong, H. Zhu (Eds.), Proc. International Conference on Formal Engineering Methods (ICFEM'10), Vol. 6447 of Lecture Notes in Computer Science, Springer, 2010, pp. 646–661.

[16] E. B. Johnsen, R. Schlatte, S. L. Tapia Tarifa, A formal model of object mobility in resource-restricted deployment scenarios, in: F. Arbab, P. Ölveczky (Eds.), Proc. 8th International Symposium on Formal Aspects of Component Software (FACS 2011), Vol. 7253 of Lecture Notes in Computer Science, Springer, 2012, pp. 185–202, to appear.

[17] J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, S. L. Tapia Tarifa, User-defined schedulers for real-time concurrent objects, Innovations in Systems and Software Engineering 9 (1) (2013) 29–43.
URL `http://dx.doi.org/10.1007/s11334-012-0184-5`

[18] B. C. Pierce, Types and Programming Languages, The MIT Press, 2002.

[19] K. G. Larsen, P. Pettersson, W. Yi, UPPAAL in a nutshell, International Journal on Software Tools for Technology Transfer 1 (1–2) (1997) 134–152.

[20] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, COSTABS: a cost and termination analyzer for ABS, in: O. Kiselyov, S. Thompson (Eds.), Proc. Workshop on Partial Evaluation and Program Manipulation (PEPM'12), ACM, 2012, pp. 151–154.

[21] P. C. Ölveczky, J. Meseguer, Semantics and pragmatics of Real-Time Maude, Higher-Order and Symbolic Computation 20 (1–2) (2007) 161–196.

[22] J. Schäfer, A. Poetzsch-Heffter, JCoBox: Generalizing active objects to concurrent components, in: European Conference on Object-Oriented Programming (ECOOP 2010), Vol. 6183 of Lecture Notes in Computer Science, Springer, 2010, pp. 275–299.

[23] A. Welc, S. Jagannathan, A. Hosking, Safe futures for Java, in: Proc. Object oriented programming, systems, languages, and applications (OOPSLA'05), ACM Press, New York, NY, USA, 2005, pp. 439–453.

[24] W. Ahrendt, M. Dylla, A system for compositional verification of asynchronous objects, Science of Computer Programming (2012), in press. `doi:10.1016/j.scico.2010.08.003`.

[25] E. Albert, S. Genaim, M. Gómez-Zamalloa, E. B. Johnsen, R. Schlatte, S. L. Tapia Tarifa, Simulating concurrent behaviors with worst-case cost bounds, in: M. Butler, W. Schulte (Eds.), FM 2011, Vol. 6664 of Lecture Notes in Computer Science, Springer, 2011, pp. 353–368.

[26] R. Schlatte, E. B. Johnsen, F. Kazemeyni, S. L. Tapia Tarifa, Models of rate restricted communication for concurrent objects, Electronic Notes in Theoretical Computer Science 274 (2011) 67–81.

[27] E. B. Johnsen, R. Schlatte, S. L. Tapia Tarifa, A formal model of user-defined resources in resource-restricted deployment scenarios, in: B. Beckert, F. Damiani, D. Gurov (Eds.), Proc. International Conference on Formal Verification of Object-Oriented Software (FoVeOOS'11), Vol. 7421 of Lecture Notes in Computer Science, Springer, 2012, pp. 196–213.

[28] E. Albert, F. de Boer, R. Hähnle, E. B. Johnsen, C. Laneve, Engineering virtualized services, in: M. A. Babar, M. Dumas (Eds.), 2nd Nordic Symposium on Cloud Computing & Internet Technologies (NordiCloud'13), ACM, 2013, pp. 59–63.

[29] A. Igarashi, N. Kobayashi, Resource usage analysis, ACM Transactions on Programming Languages and Systems 27 (2) (2005) 264–313.

[30] M. Hennessy, A Distributed Pi-Calculus, Cambridge University Press, 2007.

[31] R. D. Nicola, G. L. Ferrari, U. Montanari, R. Pugliese, E. Tuosto, A process calculus for QoS-aware applications, in: J.-M. Jacquet, G. P. Picco (Eds.), Proc. 7th International Conference on Coordination Models and Languages (COORDINATION'05), Vol. 3454 of Lecture Notes in Computer Science, Springer, 2005, pp. 33–48.

[32] F. Barbanera, M. Bugliesi, M. Dezani-Ciancaglini, V. Sassone, Space-aware ambients and processes, Theoretical Computer Science 373 (1–2) (2007) 41–69.

[33] G. Lüttgen, W. Vogler, Bisimulation on speed: A unified approach, Theoretical Computer Science 360 (1–3) (2006) 209–227.

[34] R. Jain, The Art of Computer Systems Performance Analysis, John Wiley & Sons, Inc., 1991.

[35] X. Bai, M. Li, B. Chen, W.-T. Tsai, J. Gao, Cloud testing tools, in: J. Z. Gao, X. Lu, M. Younas, H. Zhu (Eds.), Proc. 6th Intl. Symposium on Service Oriented System Engineering (SOSE'11), IEEE, 2011, pp. 1–12.

[36] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. D. Rose, R. Buyya, CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, Software, Practice and Experience 41 (1) (2011) 23–50.

[37] A. Nuñez, J. Vázquez-Poletti, A. Caminero, G. Castañé, J. Carretero, I. Llorente, iCanCloud: A flexible and scalable cloud infrastructure simulator, Journal of Grid Computing 10 (2012) 185–209.

[38] R. Buyya, M. Murshed, GridSim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing, Concurrency and Computation: Practice and Experience 14 (2002) 1175–1220.

[39] R. N. Calheiros, M. A. Netto, C. A. D. Rose, R. Buyya, EMUSIM: an integrated emulation and simulation environment for modeling, evaluation, and validation of performance of cloud computing applications, Software: Practice and Experience 43 (5) (2013) 595–612.

[40] R. N. Calheiros, R. Buyya, C. A. F. De Rose, Building an automated and self-configurable emulation testbed for grid applications, Software: Practice and Experience 40 (5) (2010) 405–429.

[41] E. Deelman, G. Singh, M. Livny, G. B. Berriman, J. Good, The cost of doing science on the cloud: The Montage example, in: Proceedings of the Conference on High Performance Computing (SC'08), IEEE/ACM, 2008, pp. 1–12.

[42] E. B. Johnsen, R. Schlatte, S. L. Tapia Tarifa, Modeling resource-aware virtualized applications for the cloud in Real-Time ABS, in: T. Aoki, K. Tagushi (Eds.), Proc. 14th International Conference on Formal Engineering Methods (ICFEM'12), Vol. 7635 of Lecture Notes in Computer Science, Springer, 2012, pp. 71–86.

[43] F. S. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte, P. Y. H. Wong, Formal modeling of resource management for cloud architectures: An industrial case study, in: F. D. Paoli, E. Pimentel, G. Zavattaro (Eds.), Proc. European Conference on Service-Oriented and Cloud Computing (ESOCC 2012), Vol. 7592 of Lecture Notes in Computer Science, Springer, 2012, pp. 91–106.

[44] I. Epifani, C. Ghezzi, R. Mirandola, G. Tamburrelli, Model evolution by run-time parameter adaptation, in: Proc. 31st International Conference on Software Engineering (ICSE'09), IEEE, 2009, pp. 111–121.

[45] S. Balsamo, A. D. Marco, P. Inverardi, M. Simeoni, Model-based performance prediction in software development: A survey, IEEE Transactions on Software Engineering 30 (5) (2004) 295–310.

[46] A. Vulgarakis, C. C. Seceleanu, Embedded systems resources: Views on modeling and analysis, in: Proc. 32nd IEEE Intl. Computer Software and Applications Conference (COMPSAC'08), IEEE Computer Society, 2008, pp. 1321–1328.

[47] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, M. Stoelinga, Resource interfaces, in: R. Alur, I. Lee (Eds.), Proc. Third International Conference on Embedded Software (EMSOFT'03), Vol. 2855 of Lecture Notes in Computer Science, Springer, 2003, pp. 117–133.

[48] E. Fersman, P. Krcál, P. Pettersson, W. Yi, Task automata: Schedulability, decidability and undecidability, Information and Computation 205 (8) (2007) 1149–1172.

[49] M. M. Jaghoori, F. S. de Boer, T. Chothia, M. Sirjani, Schedulability of asynchronous real-time concurrent objects, Journal of Logic and Algebraic Programming 78 (5) (2009) 402–416.

[50] F. S. de Boer, M. M. Jaghoori, E. B. Johnsen, Dating concurrent objects: Real-time modeling and schedulability analysis, in: P. Gastin, F. Laroussinie (Eds.), Proc. 21st Intl. Conf. on Concurrency Theory (CONCUR), Vol. 6269 of Lecture Notes in Computer Science, Springer, 2010, pp. 1–18.

[51] H. Koziolek, Performance evaluation of component-based software systems: A survey, Performance Evaluation 67 (8) (2010) 634–658.

[52] C. U. Smith, L. G. Williams, Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software, Addison-Wesley, 2002.

[53] D. B. Petriu, C. M. Woodside, An intermediate metamodel with scenarios and resources for generating performance models from UML designs, Software and System Modeling 6 (2) (2007) 163–184.

[54] M. Verhoef, P. G. Larsen, J. Hooman, Modeling and validating distributed embedded real-time systems with VDM++, in: J. Misra, T. Nipkow, E. Sekerinski (Eds.), Proceedings of the 14th International Symposium on Formal Methods (FM'06), Vol. 4085 of Lecture Notes in Computer Science, Springer, 2006, pp. 147–162.

[55] R. Reussner, H. W. Schmidt, I. Poernomo, Reliability prediction for component-based software architectures, Journal of Systems and Software 66 (3) (2003) 241–252.

[56] S. Becker, H. Koziolek, R. Reussner, The Palladio component model for model-driven performance prediction, Journal of Systems and Software 82 (1) (2009) 3–22.

[57] L. Happe, B. Buhnova, R. Reussner, Stateful component-based performance models, Journal of Software and Systems Modeling. Available online: `http://dx.doi.org/10.1007/s10270-013-0336-6`. To appear.

[58] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, Cost Analysis of Java Bytecode, in: 16th European Symposium on Programming, (ESOP'07), Vol. 4421 of Lecture Notes in Computer Science, Springer, 2007, pp. 157–172.

[59] S. Gulwani, K. K. Mehra, T. M. Chilimbi, SPEED: Precise and Efficient Static Estimation of Program Computational Complexity, in: Z. Shao, B. C. Pierce (Eds.), Proc. 36th Symp. on Principles of Programming Languages (POPL'09), ACM, 2009, pp. 127–139.

[60] B. Beckert, R. Hähnle, P. H. Schmitt (Eds.), Verification of Object-Oriented Software. The KeY Approach, Vol. 4334 of Lecture Notes in Artificial Intelligence, Springer, 2007.

[61] K. Sen, M. Viswanathan, G. Agha, On statistical model checking of stochastic systems, in: K. Etessami, S. K. Rajamani (Eds.), Proc. 17th International Conference on Computer Aided Verification (CAV'05), Vol. 3576 of Lecture Notes in Computer Science, Springer, 2005, pp. 266–280.

[62] A. David, K. G. Larsen, A. Legay, M. Mikucionis, D. B. Poulsen, J. van Vliet, Z. Wang, Statistical model checking for networks of priced timed automata, in: U. Fahrenberg, S. Tripakis (Eds.), Proc. 9th Intl. Conf. on Formal modeling and analysis of timed systems (FORMATS'11), Vol. 6919 of Lecture Notes in Computer Science, Springer, 2011, pp. 80–96.

[63] M. AlTurki, J. Meseguer, PVeStA: A parallel statistical model checking and quantitative analysis tool, in: A. Corradini, B. Klin, C. Cîrstea (Eds.), Proc. 4th International Conference on Algebra and Coalgebra in Computer Science

(CALCO'11), Vol. 6859 of Lecture Notes in Computer Science, Springer, 2011, pp. 386–392.

CHAPTER 7

# Paper 3: Modeling Resource-Aware Virtualized Applications for the Cloud *

**_Authors:_** Einar Broch Johnsen, Rudolf Schlatte, Silvia Lizeth Tapia Tarifa.

**_Publication:_** Formal Methods and Software Engineering. Proceedings of the 14th International Conference on Formal Engineering Methods, volume 7635 of Lecture Notes in Computer Science, pages 71–86. Springer, November 2012.

**_Abstract:_** An application's quality of service (QoS) depends on resource availability; e.g., response time is worse on a slow machine. On the cloud, a virtualized application leases resources which are made available on demand. When its work load increases, the application must decide whether to reduce QoS or increase cost. Virtualized applications need to manage their acquisition of resources. In this paper resource provisioning is integrated in high-level models of virtualized applications. We develop a Real-Time ABS model of a cloud provider which leases virtual machines to an application on demand. A case study of the Montage system then demonstrates how to use such a model to compare resource management strategies for virtualized software during software design. Real-Time ABS is a timed abstract behavioral specification language targeting distributed object-oriented systems, in which dynamic deployment scenarios can be expressed in executable models.

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

## 7.1 Introduction

The added value and compelling business drivers of cloud computing are undeniable [10], but considerable new challenges need to be addressed for industry to make an effective usage of cloud computing. As the key technology enabler for cloud computing, *virtualization* makes elastic amounts of resources available to application-level services deployed on the cloud; for example, the processing capacity allocated to a service may be changed on the demand. The integration of virtualization in general purpose software applications requires novel techniques for leveraging resources and resource management into software engineering. Virtualization poses challenges for the software-as-a-service abstraction concerning the development, analysis, and dynamic composition of software with respect to quality of service. Today these challenges are not satisfactorily addressed in software engineering. In particular, better support for the modeling and validation of application-level resource management strategies for virtualized resources are needed to help the software developer make efficient use of the available virtualized resources in their applications.

The abstract behavioral specification language ABS is a formalism which aims at describing systems at a level which abstracts from many implementation details but captures essential behavioral aspects of the targeted systems [25]. ABS targets the engineering of concurrent, component-based systems by means of executable object-oriented models which are easy to understand for the software developer and allow rapid prototyping and analysis. The extension Real-Time ABS integrates object orientation and timed behavior [8]. Whereas the functional correctness of a planned system largely depends on its high-level behavioral specification, the choice of deployment architecture may hugely influence the system's quality of service. For example, CPU limitations may restrict the applications that can be supported on a cell phone, and the capacity of a server may influence the response time of a service during peaks in the user traffic.

Whereas software components reflect the logical architecture of systems, *deployment components* have recently been proposed for Real-Time ABS to reflect the deployment architecture of systems [27, 28]. A deployment component is a resource-restricted execution context for a set of concurrent object groups, which controls how much computation can occur in this set between observable points in time. Deployment components may be dynamically created and are parametric in the amount of resources they provide to their objects. This explicit representation of deployment scenarios allows application-level response time and load balancing to be expressed in the software models in a very natural and flexible way, relative to the resources allocated to the software.

This paper shows how deployment components in Real-Time ABS may be used to model virtualized systems in a cloud environment. We develop a Real-Time ABS

model of cloud provisioning and accounting for resource-aware applications: an abstract cloud provider offers virtual machines with given CPU capacities to client applications and bills the applications for their resource usage. We use this model in a case study of the Montage system [24], a cloud-based resource-aware application for scientific computing, and compare execution times and accumulated costs depending on the number of leased machines by means of simulations of the executable model. We show that our results are comparable to those previously obtained for Montage with the same deployment scenarios on specialized simulation tools [19] and thus that our formal model can be used to estimate cloud deployment costs for realistic systems. We then introduce dynamic resource management strategies in the Montage model, and show that these improve on the resource management strategies previously considered [19].

The paper is structured as follows. Section 7.2 presents the abstract behavioral specification language Real-Time ABS, Section 7.3 develops our model of cloud provisioning. Section 7.4 presents the case study of the Montage system. Section 7.5 discusses related work and Section 7.6 concludes the paper.

## 7.2 Abstract Behavioral Specification with Real-Time ABS

ABS is an executable object-oriented modeling language with a formal semantics [25], which targets distributed systems. The language is based on concurrent object groups, akin to concurrent objects (e.g., [14, 17, 26]), Actors (e.g., [1, 23]), and Erlang processes [5]. Concurrent object groups in ABS internally support interleaved concurrency using guarded commands. This allows active and reactive behavior to be easily combined, based on cooperative scheduling of processes which stem from method calls. A concurrent object group has at most one active process at any time and a queue of suspended processes waiting to execute on an object in the group. Objects in ABS are dynamically created from classes but typed by interface; i.e., there is no explicit notion of hiding as the object state is always encapsulated behind interfaces which offer methods to the environment.

### 7.2.1 Modeling Timed Behavior in ABS

ABS combines functional and imperative programming styles with a Java-like syntax [25]. Concurrent object groups execute in parallel and communicate through asynchronous method calls. Data manipulation inside methods is modeled using a simple functional language. Thus, the modeler may abstract from the details of low-level imperative implementations of data structures, and still maintain an overall

object-oriented design which is close to the target system.

The *functional* part of ABS allows user-defined algebraic data types such as the empty type Unit, Booleans Bool, integers Int; parametric data types such as sets Set<A> and maps Map<A> (given a value for the variable A); and user-defined functions over values of these types, with support for pattern matching.

The *imperative* part of ABS addresses concurrency, communication, and synchronization at the concurrent object level, and defines interfaces, classes, and methods. ABS objects are *active* in the sense that their run method, if defined, gets called upon creation. *Statements* for sequential composition $s_1; s_2$, assignment x=rhs, **skip**, **if**, **while**, and **return** are standard. The statement **suspend** unconditionally suspends the active process of an object by moving this process to the queue, from which an enabled process is selected for execution. In **await** $g$, the guard $g$ controls suspension of the active process and consists of Boolean conditions $b$ and return tests $x$? (see below). Functional expressions $e$ and guards $g$ are side-effect free. If $g$ evaluates to false, the active process is suspended, i.e., moved to the queue, and some process from the queue may execute. *Expressions* rhs include the creation of an object group **new cog** C(e), object creation in the creator's group **new** C(e), method calls `o!m(e)` and o.m(e), future dereferencing x.**get**, and functional expressions e.

*Communication* and *synchronization* are decoupled in ABS, which allows complex workflows to be modeled. Communication is based on asynchronous method calls, denoted by assignments `f=o!m(e)` where f is a future variable, o an object expression, and e are (data value or object) expressions. After calling `f=o!m(e)`, the future variable f refers to the return value of the call and the caller may proceed with its execution *without blocking* on the method reply. There are two operations on future variables, which control synchronization in ABS. First, the statement **await** f? *suspends the active process* unless a return value from the call associated with f has arrived, allowing other processes in the object group to execute. Second, the return value is retrieved by the expression f.**get**, which *blocks all execution in the object* until the return value is available. The statement sequence `x=o!m(e);v=`x.**get** encodes commonly used *blocking calls*, abbreviated v=o.m(e) (reminiscent of synchronous calls).

We work with Real-Time ABS [8], a timed extension of ABS with a run-to-completion semantics, which combines *explicit* and *implicit* time for ABS models. Real-Time ABS has an interpreter defined in rewriting logic [30] which closely reflects its semantics and which executes on the Maude platform [16]. In Real-Time ABS, explicit time is specified directly in terms of durations (as in, e.g., UPPAAL [29]). Real-Time ABS provides the statement **duration**(b,w) to specify a duration between the worst-case w and the best case b. A process may also suspend for a certain duration, expressed by **await duration**(b,w). For the purposes of this paper, it is sufficient to work with a discrete time domain, and let b and w be of type Int. In contrast to explicit time, implicit time is *observed* by measurements of the executing model. Measurements are obtained by comparing clock values from a global clock, which can be

read by an expression **now**() of type Time. With implicit time, no assumptions about execution times are hard-coded into the models. The execution time of a method call depends on how quickly the call is effectuated by the server object. In fact, the execution time of a statement varies with the *capacity* of the chosen deployment architecture and on *synchronization* with other (slower) objects.

### *7.2.2 Modeling Deployment Architectures in Real-Time ABS*

*Deployment components* in Real-Time ABS abstractly capture the resource capacity at a location [27, 28]. Deployment components are first-class citizens in Real-Time ABS and share their resources between their allocated objects. The root object of a model is allocated to the deployment component environment, which has unlimited resources. Deployment components with different resource capacities may be dynamically created depending on the control flow of the model or statically created in the main block of the model. When created, objects are by default allocated to the same deployment component as their creator, but they may also be explicitly allocated to a different component by an annotation.

Deployment components have the type DC and are instances of the class Deployment Component. This class takes as parameters a name (the name of the location, mostly used for monitoring purposes), given as a string, and a set of restrictions on resources. Here we focus on resources reflecting the components' *CPU processing* capacity, which are specified by the constructor CPUCapacity(r), where r of type Resource represents the amount of available abstract processing resources between observable points in time. The expression **thisDC**() evaluates to the deployment component of the current object. The method **total**("CPU") of a deployment component returns the total amount of CPU resources allocated to that component.

The *CPU processing capacity* of a deployment component determines how much computation may occur in the objects allocated to that component. The CPU resources of a component define its capacity between observable (discrete) points in time, after which the resources are renewed. Objects allocated to the component compete for the shared resources in order to execute. With the run-to-completion semantics, the objects may execute until the component runs out of resources or they are otherwise blocked, after which time will advance [28].

The *cost* of executing statements is given by a cost model. A default cost value for statements can be set as a compiler option (e.g., defaultcost=10). This default cost does not discriminate between different statements. For some statements a more precise cost expression is desirable in a realistic model; e.g., if e is a complex expression, then the statement x=e should have a significantly higher cost than the statement **skip**. For this reason, more fine-grained costs can be introduced into the models by means

of annotations, as follows:

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

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

## 7.3 Resource Management and Cloud Provisioning

An explicit model of cloud provisioning allows the application developer to interact in a simple way with a provisioning and accounting system for virtual machines. This section explains how such cloud provisioning may be modeled, for Infrastructure-as-a-Service [10] cloud environments. Consider an interface CloudProvider which offers three methods for resource management to client applications: createMachine, acquireMachine, and releaseMachine.

The method createMachine prepares and returns an abstract virtual machine with a specified processing capacity, after which the client application may deploy objects on the machine. This method models the provisioning and configuration part of a cloud-based application, and corresponds roughly to instancing and configuring a virtual machine on a cloud, without starting up the machine.

Before running a computation on a machine created with createMachine, the client application must first call the method acquireMachine. The cloud provider then starts *accounting* for the time this machine is kept running; the client calls the method releaseMachine to "shut down" the machine again. (For simplicity it is currently not checked whether processes are run before calling acquireMachine or after releaseMachine; this is a straightforward extension of the approach which could be useful to model "cheating" clients.) For a later reactivation of the same machine, only

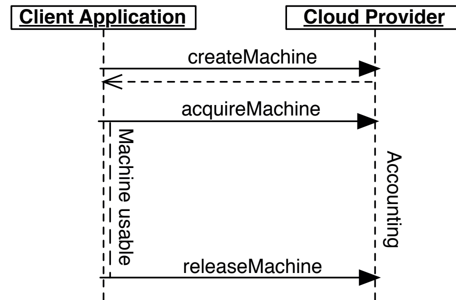


Figure 7.1: Interaction between a client application and the cloud provider.

acquireMachine needs to be called. Fig. 7.1 shows one such sequence of interactions between a client application and a cloud provider.

In addition, the interface offers a method getAccumulatedCost which returns the cost accumulated so far by the client application. This method can be used in load balancing schemes to implement various trade-offs between quality of service and the cost of running the application, or to implement operator alerts when certain QoS or cost budgets are bypassed.

***A Model of Cloud Provisioning in Real-Time ABS.*** A class which implements the CloudProvider interface is given in Fig. 7.2. Abstract virtual machines are modeled as deployment components. The class has two formal parameters to allow easy configuration: startupTime sets the length of the startup procedure for virtual machines and accountingPeriod sets the length of each accounting period. In addition, the class has four fields: accumulatedCost stores the cost incurred by the client application up to present time, the set billableMachines contains the machines to be billed in the current time interval, and the sets availableMachines and runningMachines contain the created but not currently running and the running machines, respectively. The empty set is denoted EmptySet. Let $s$ be a set over elements of type $T$ and let $e : T$. The following functions are defined in the functional part of Real-Time ABS: insertElement$(s, e)$ returns $\{e\} \cup s$, remove$(s, e)$ returns $s \setminus \{e\}$, and take$(s)$ returns some $e$ such that $e \in s$.

The methods for resource management move machines between these sets. Any machine which is either created or running within an accounting period, is billable in that period; i.e., a machine may be both acquired and released in a period, so there may be more billable than running machines. The method createMachine creates a new deployment component of the given capacity and adds it to availableMachines. The method acquireMachine moves a machine from availableMachines to runningMachines.

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

class CloudProvider (Int startupTime, Int accountingPeriod) implements CloudProvider {
  Int accumulatedCost = 0;
  Set<DC> billableMachines = EmptySet;
  Set<DC> availableMachines = EmptySet;
  Set<DC> runningMachines = EmptySet;

  DC createMachine(Int r) {
    DC dc = new DeploymentComponent("", set[CPUCapacity(r)]);
    availableMachines = insertElement(availableMachines, dc);
    return dc;
  }

  Unit acquireMachine(DC dc) {
    billableMachines = insertElement(billableMachines, dc);
    availableMachines = remove(availableMachines, dc);
    runningMachines = insertElement(runningMachines, dc);
    await duration(startupTime, startupTime);
  }

  Unit releaseMachine(DC dc) {
    runningMachines = remove(runningMachines, dc);
    availableMachines = insertElement(availableMachines, dc);
  }

  Int getAccumulatedCost(){
    return accumulatedCost;
  }

  Unit run() {
    while (True) {
      await duration(accountingPeriod, accountingPeriod);
      Set<DeploymentComponent> billables = billableMachines;
      while (~(billables == EmptySet)) {
        DeploymentComponent dc = take(billables);
        billables = remove(billables,dc);
        Int capacity = dc.total("CPU");
        accumulatedCost = accumulatedCost+(accountingPeriod*capacity);
      }
      billableMachines = runningMachines;
    }
  }
}
```

Figure 7.2: The CloudProvider class in Real-Time ABS.

Since the machine becomes billable, it is placed in billableMachines. The method suspends for the duration of the startupTime before it returns, so the accounting includes the startup time of the machine. The method releaseMachine moves a machine from runningMachines to availableMachines. The machine remains billable for the current accounting period.

The run method of the cloud provider implements the accounting of incurred resource usage for the client application. The method suspends for the duration of the accounting period, after which all machines in billableMachines are billed by adding their resource capacity for the duration of the accounting period to accumulatedCost. Remark that Real-Time ABS has a run-to-completion semantics which guarantees that the loop in run will be executed after every accounting period. After accounting is finished, only the currently running machines are already billable for the next period. These are copied into billableMachines and the run method suspends for the next accounting period.

## 7.4 Case Study: The Montage Toolkit

Montage is a portable software toolkit for generating science-grade mosaics by composing multiple astronomical images [24]. Montage is modular and can be run on a researcher's desktop machine, in a grid, or on a cloud. Due to the high volume of data in a typical astronomical dataset and the high resolution of the resulting mosaic, as well as the highly parallelizable nature of the needed computations, Montage is a good candidate for cloud deployment. In [19], Deelman et al. present simulations of cloud deployments of Montage and the cost of creating mosaics with different deployment scenarios, using the specialized simulation tool GridSim [9].

This section describes the architecture of the Montage system and how it was modeled in Real-Time ABS. We explain how costs were associated to the different parts of the model. The results obtained by simulations of the model in the Real-Time ABS interpreter are compared to those obtained in the specialized simulator. Finally, more fine-grained dynamic resource management, not considered in the previous work [19], is proposed and compared to previous scenarios.

### 7.4.1 The Problem Description

Creating a mosaic from a set of input images involves a number of tasks: first reprojecting the images to a common projection, coordinating system and scale, then rectifying the background radiation in all images to a common flux scale and background level, and finally co-adding the reprojected background-rectified images into a final mosaic. The tasks exchange data in the format FITS, which encapsulates

| Module | Description |
|---|---|
| **mImgtbl** | Extract geometry information from a set of FITS headers and create a metadata table from it. |
| **mOverlaps** | Analyze an image metadata table to determine which images overlap on the sky. |
| **mProject** | Reproject a FITS image. |
| **mProjExec** | Reproject a set of images, running *mProject* for each image. |
| **mDiff** | Perform a simple image difference between a pair of overlapping images. |
| **mDiffExec** | Run *mDiff* on all the overlap pairs identified by *mOverlaps*. |
| **mFitplane** | Fit a plane (excluding outlier pixels) to an image. Used on the difference images generated by *mDiff*. |
| **mFitExec** | Run *mFitplane* on all overlapping pairs. Creates a table of image-to-image difference parameters. |
| **mBgModel** | Modeling/fitting program which uses the image-to-image difference parameter table to interactively determine a set of corrections to apply to each image to achieve a "best" global fit. |
| **mBackground** | Remove a background from a single image |
| **mBgExec** | Run *mBackground* on all the images in the metadata table. |
| **mAdd** | Co-add the reprojected images to produce an output mosaic. |

Figure 7.3: The modules of the Montage case study.

image data and meta-data. These tasks are implemented by a number of Montage modules [24], which are listed and described in Fig. 7.3. These modules can be run individually or combined in a workflow, locally or remotely on a grid or a cloud. Fig. 7.4 depicts the dataflow dependencies between the modules in a typical Montage workflow [19]. These dependencies show which jobs can be parallelized on multiprocessor systems, grids, or cloud services.

Simulation results for running Montage on the *Amazon* cloud with the workflow depicted in Fig. 7.4 have been published in [19], including cost measurements for CPU and storage resources. The simulation tool GridSim [9] was used to study the trade-offs between cost and performance for different execution and resource provisioning scenarios when running Montage in a cloud service provider.

We model and analyze the same abstract workflow architecture of Montage based on the model of cloud provisioning presented in Section 7.3, as a means to validate the presented formal model of cloud provisioning in Real-Time ABS. In particular, we consider the case in which Montage processes multiple input images in parallel. Our model abstracts from the implementation details of the manipulation of images, replacing them with abstract statements and cost annotations. One important result of [19] is that computation cost dominates storage and data transfer cost for the

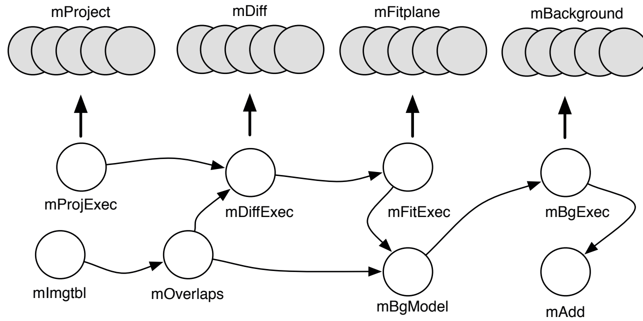


Figure 7.4: Montage abstract workflow.

Montage workload by 2-3 orders of magnitude, which allows us to focus on CPU usage alone.

### 7.4.2 A Model of the Montage Workflow in Real-Time ABS

***The Core Modules.*** The Montage core modules that execute atomic tasks (i.e., mProject, mDiff, mFitplane, mBgModel, mBackground, mAdd, mImgtbl, and mOverlaps) are modeled as methods inside a class CalcServer which implements the CalcServer interface shown in Fig. 7.5. In the methods of this class, cost annotations are used to specify the costs of executing atomic tasks. The images considered in the case study have a constant size, so it is sufficient to use a constant cost for the atomic tasks. Lacking precise cost estimates for the individual tasks, we consider an abstract cost model in which each atomic task is assigned the cost of 1 resource. (This cost model could be further refined; although some timing measurements are given in [24], these are not detailed enough for this purpose.) The code for one such atomic task inside the CalcServer class is shown in Fig. 7.5.

***Resource Management.*** The workflow process does not interact with the different instances of CalcServer directly. Instead, tasks are sent to an instance of ApplicationServer which acts a broker for the preallocated machine instances and distributes tasks to free machines. The ApplicationServer interface, partly shown in Fig. 7.6, provides the workflow with means to start the parallelizable tasks (i.e., mProjExec, mDiffExec, mFitExec and mBgExec) and distributes the atomic tasks (e.g., mDiff) to instances of CalcServer. Atomic tasks are sent directly to one calculation server. Two fields activeMachines and servers keep track of the number of active jobs on

```
interface CalcServer {
  DeploymentComponent getDC();
  MetadataT mImgtbl(List<FITS> i);
  MetadataT mOverlaps(MetadataT mt);
  FITS mProject(FITS image);
  FITSdf mDiff (FITS image1, FITS image2);
  FITSfit mFitplane (FITSdf df);
  CorrectionT mBgModel(Image2ImageT diffs, MetadataT ovlaps);
  FITS mBackground (Int correction,FITS image );
  FITS mAdd (List<FITS> images); }

class CalcServer implements CalcServer {
  ...
  FITS mBackground (Int correction,FITS image ){
    [Cost: 1] FITS result = correctFITS(image,correction);
    return result;
  }
  ...
}
```

Figure 7.5: CalcServer interface and class in Real-Time ABS.

each created machine and the order in which servers get jobs, respectively. Surrounding every call to a calculation server the auxiliary methods getServer and dropServer do the bookkeeping and resource management of the virtual machines. Asynchronous method calls to the future variables fimage and fnewimages, and task suspension are used to keep the application server responsive.

Our model defines algebraic data types FITS, FITSdf, FITSfit, as well as the list MetadataT and the maps CorrectionT and Image2ImageT to represent the input and output data at the different stages of the workflow; for example, FITS is a data type which represents image archives in FITS format, which is constructed from an abstract representation of metadata and of image data. This data can be used to keep track of data flow and abstractions of calculation results. The empty list and map are denoted Nil and EmptyMap. On lists, the constructor Cons$(h, t)$ takes as arguments an element $h$ and a list $t$; head(Cons$(h, t)$) $= h$ and tail(Cons$(h, t)$) $= t$. The function isEmpty$(l)$ returns true if $l$ is the empty list. On maps, the function lookupDefault$(m, k, v)$ returns the value bound to $k$ in $m$ if the key $k$ is bound in $m$, and otherwise it returns the default value $v$.

### *7.4.3 Simulation Results*

We simulated a workload equivalent to the *Montage 1* scenario described in [19]. As in that paper, the simulations were run on deployment scenarios ranging from 1 to 128 virtual machine instances, where all the machines were started up prior to the simulations (i.e., the startupTime parameter of the CloudProvider class in our model

```
interface ApplicationServer {
  FITS mAdd (List<FITS> images);
  List<FITS> mProjExec(List<FITS> images);
  List<FITSdf> mDiffExec (MetadataT metatable, List<FITS> images);
  Image2ImageT mFitExec(List<FITSdf> dfs);
  List<FITS> mBgExec (CorrectionT corrections, List<FITS> images);
  ...
}

class ApplicationServer(CloudProvider provider) implements ApplicationServer {
  List<CalcServer> servers = Nil; Map<DC,Int> activeMachines = EmptyMap;
  ...
  List<FITS> mBgExec(CorrectionT corrections,List<FITS> images) {
    List<FITS> newimages = Nil;
    if (isEmpty(images)==False) {
      FITS image = head(images);
      Int correction = lookupDefault(corrections,getId(image), 0);
      CalcServer b = this.getServer();
      Fut<FITS> fimage = b!mBackground (correction,image);
      Fut<List<FITS>> fnewimages=this!mBgExec(corrections,tail(images));
      await fimage?;
      FITS tmpimage = fimage.get;
      this.dropServer(b);
      await fnewimages?;
      List<FITS> newtmpimages = fnewimages.get;
      newimages = Cons(tmpimage, newtmpimages);
    }
    return newimages;
  }
  ...
}
```

Figure 7.6: The ApplicationServer interface and class (abridged).

has value 0). Both simulation approaches exhibit the expected geometric downward progression of execution time when going from 1 to 128 machines, and roughly half an order of magnitude increase in cost. In our first simulation runs, the execution cost (measured in simulated machine-minutes) increased a little over two-fold over the full simulation range, versus closer to a six-fold increase (“60 cents [...] versus almost 4$”) in [19]. To explain this difference, we theorized that the observed lower cost may have resulted from better machine allocation strategies in our model—the virtual machines were eagerly released by the ApplicationServer class when no more work was available to them, instead of being kept running until all computations finished.

To test this hypothesis, the ApplicationServer class was modified to keep all instances running during the whole computation task. Using this allocation strategy, we observed a cost increase of 4.27 from 1 to 128 computation servers, which is more in line with the results obtained using GridSim. Fig. 7.7 (left) shows the simulation results of the modified model. The authors of [19] later confirmed in private commu-

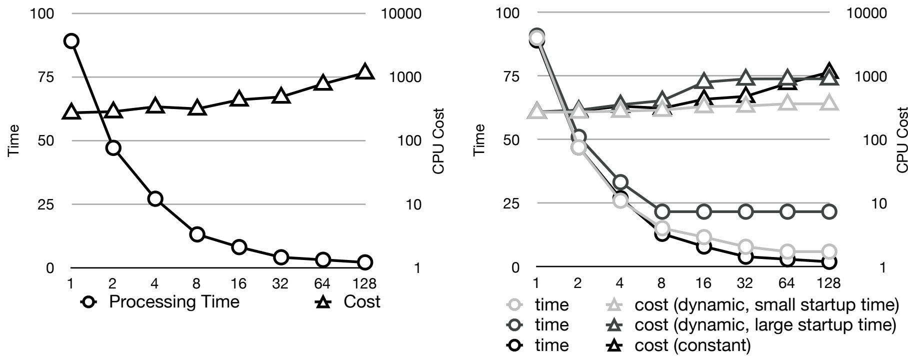


Figure 7.7: Execution costs and times of simulation. The *Montage 1* scenario (left figure) is compared to dynamic resource management (right figure). The costs are presented on a logarithmic scale for easier comparison with the results of [19].

nication that our hypothesis about the setup of the GridSim simulation scenario was indeed correct.

In order to further investigate the initial results involving dynamic startup and shutdown of machine instances, we refine our model by introducing startup times for virtual machines. Fig. 7.7 (right) compares the previous static deployment scenario (constant) with two dynamic resource management scenarios with varying startup times for virtual machines. One scenario models machine startup times of roughly one tenth of the time needed for performing a basic task, the other startup times roughly as large as basic task times. It can be seen that the cost of running a single job in the Montage system can be substantially reduced by switching off unused machines, given that the cost of starting machines is dominated by the actual calculations taking place, with almost no loss in time. On the other hand, if starting a machine is significantly slower than executing a basic task, it can be seen that both cost and time of the dynamic scenario are worse than when initially starting all machines in the static scenario of the considered workflow except in the case of severe over-provisioning of machines.

## 7.5 Related Work

The concurrency model of ABS is based on concurrent objects and Actor-based computation, in which software units with encapsulated processors communicate asynchronously (e.g., [1, 5, 14, 23, 26]). Their inherent compositionality allows concurrent objects to be naturally distributed on different locations, because only the local state

of a concurrent object is needed to execute its methods. In previous work, the authors have introduced *deployment components* as a formal modeling concept to capture restricted resources shared between concurrent object groups and shown how components with parametric resources naturally model different deployment architectures [28], extended the approach with resource reallocation [27], and combined it with static cost analysis [4]. This paper complements our previous work by using deployment components to model cloud-based scenarios and the development of the Montage case study. A companion paper [18] further applies the approach of this paper to an industrial case study.

Techniques for prediction or analysis of non-functional properties are based on either *measurement* or *modeling*. Measurement-based approaches apply to existing implementations, using dedicated profiling or tracing tools like JMeter or LoadRunner. Model-based approaches allow abstraction from specific system intricacies, but depend on parameters provided by domain experts [20]. A survey of model-based performance analysis techniques is given in [7]. Formal systems using process algebra, Petri Nets, game theory, and timed automata have been used in the embedded software domain (e.g., [15, 21]). Real-Time ABS combines *explicit* time modeling with duration statements with *implicit* measurements of time already at the modeling level, which is made possible by the combination of costs in the application model and capacities in the deployment components.

Work on modeling object-oriented systems with resource constraints is more scarce. Using the UML SPT profile for schedulability, performance, and time, Petriu and Woodside [32] informally define the Core Scenario Model (CSM) to solve questions that arise in performance model building. CSM has a notion of resource context, which reflects an operation's set of resources. CSM aims to bridge the gap between UML and techniques to generate performance models [7]. Closer to our work is M. Verhoef's extension of VDM++ for embedded real-time systems [33], in which static architectures are explicitly modeled using CPUs and buses. The approach uses fixed resources targeting the embedded domain, namely processor cycles bound to the CPUs, while we consider more general resources for arbitrary software. Verhoef's approach is also based on abstract executable modeling, but the underlying object models and operational semantics differ. VDM++ has multi-thread concurrency, preemptive scheduling, and a strict separation of synchronous method calls and asynchronous signals, in contrast to our work with concurrent objects, cooperative scheduling, and caller-decided synchronization.

Related work on simulation tools for cloud computing are typically reminiscent of network simulators. A number of testing techniques and tools for cloud-based software systems are surveyed in [6]. In particular, CloudSim [13] and ICanCloud [31] are simulation tools using virtual machines to simulate cloud environments. CloudSim is a fairly mature tool which has already been used for a number of papers, but it is restricted to simulations on a single computer. In contrast, ICanCloud supports dis-

tribution on a cluster. Additionally CloudSim was originally based on GridSim [9], a toolkit for modeling and simulations of heterogeneous Grid resources. EMUSIM [12] is an integrated tool that uses AEF [11] (Automated Emulation Framework) to estimate performance and costs for an application by means of emulations to produce improved input parameters for simulations in CloudSim. Compared to these approaches, our work is based on a formal semantics and aims to support the developer of software applications for cloud-based environments at an early phase in the development process.

Another interesting line of research is static cost analysis for object-oriented programs (e.g., [3, 22]) Most tools for cost analysis only consider sequential programs, and assume that the program is fully developed before cost analysis can be applied. COSTABS [2] is a cost analysis tool for ABS which supports concurrent object-oriented programs. Our approach, in which the modeler specifies cost in cost annotations, could be supported by COSTABS to automatically derive cost annotations for the parts of a model that are fully implemented. In collaboration with Albert *et al.*, we have applied this approach for memory analysis of ABS models [4]. However, the full integration of COSTABS in our tool chain and the software development process remain future work.

## 7.6 Conclusion

This paper develops a model in Real-Time ABS of a cloud provider which offers virtual machines with given CPU capacities to a client application. Virtual machines are modeled as deployment components with given CPU capacities, and the cloud provider offers methods for resource management of virtual machines to client applications. The proposed model has been validated by means of a case study of the Montage toolkit, in which a typical Montage workflow was formalized. This formalization allows different user scenarios and deployment models to easily expressed and compared by means of simulations using the Real-Time ABS interpreter. The results from these simulations were comparable to those obtained for the Montage case study using specialized simulators, which suggests that models using abstract behavioral specification languages such as Real-Time ABS can be used to estimate cloud deployment costs for realistic systems.

Real-Time ABS aims to support the developer of client applications for cloud-based deployment, and in particular to facilitate the development of strategies for virtualized resource management at early stages in the development process. We are not aware of similar work addressing the formal modeling of virtualized resource management and cloud computing from the client application perspective. With the increasing focus on cloud-based deployment of general purpose software, such support could become very useful for software developers.

This paper focused on the formalization of cloud provisioning and simulations of the executable model. The presented work can be extended in a number of directions. In particular, we are interested in how to combine different virtualized resources in the same model to estimate combined costs of, e.g., computations, storage, bandwidth, and power consumption. Another extension is to strengthen the tool-based analysis support for Real-Time ABS. An integration with cost analysis tools such as COSTABS would assist the developer in providing cost annotations in the model. Furthermore, we plan to investigate symbolic execution techniques for Real-Time ABS, which would allow stronger automated analysis results than those considered here. Finally, an integration of QoS contracts with the interfaces of Real-Time ABS could form a basis for analysis abstract behavioral specifications with respect to service-level agreements.

**Acknowledgment.** We thank G. Bruce Berriman and Ewa Deelman for helping us with additional details of the Montage case study.

# Bibliography


[1] G. A. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems*. The MIT Press, Cambridge, Mass., 1986.

[2] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. COSTABS: a cost and termination analyzer for ABS. In *Proc. Workshop on Partial Evaluation and Program Manipulation (PEPM'12)*, pages 151–154. ACM, 2012.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *Proc. ESOP'07*, *LNCS* 4421, pages 157–172. Springer, 2007.

[4] E. Albert, S. Genaim, M. Gómez-Zamalloa, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Simulating concurrent behaviors with worst-case cost bounds. In *Proc. Formal Methods (FM'11)*, *LNCS* 6664, pages 353–368. Springer, June 2011.

[5] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[6] X. Bai, M. Li, B. Chen, W.-T. Tsai, and J. Gao. Cloud testing tools. In *Proc. 6th Symposium on Service Oriented System Engineering*, pages 1–12. IEEE, 2011.

[7] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[8] J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 2012. `http://dx.doi.org/10.1007/s11334-012-0184-5`

[9] R. Buyya and M. Murshed. GridSim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and Computation: Practice and Experience*, 14:1175–1220, 2002.

[10] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599–616, 2009.

[11] R. N. Calheiros, R. Buyya, and C. A. F. De Rose. Building an automated and self-configurable emulation testbed for grid applications. *Software: Practice and Experience*, 40(5):405–429, Apr. 2010.

[12] R. N. Calheiros, M. A. Netto, C. A. D. Rose, and R. Buyya. EMUSIM: an integrated emulation and simulation environment for modeling, evaluation, and validation of performance of cloud computing applications. *Software: Practice and Experience*, pages 00–00, 2012.

[13] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. D. Rose, and R. Buyya. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software, Practice and Experience*, 41(1):23–50, 2011.

[14] D. Caromel and L. Henrio. *A Theory of Distributed Objects*. Springer, 2005.

[15] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Resource interfaces. In *Proc. EMSOFT'03*, *LNCS* 2855, pages 117–133. Springer, 2003.

[16] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude*, *LNCS* 4350. Springer, 2007.

[17] F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *Proc. ESOP'07*, *LNCS* 4421, pages 316–330. Springer, 2007.

[18] F. S. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte, and P. Y. H. Wong. Formal Modeling of Resource Management for Cloud Architectures: An Industrial Case Study. In *Proc. European Conference on Service-Oriented and Cloud Computing (ESOCC)*, To appear in *LNCS*. Springer, Sep. 2012.

[19] E. Deelman, G. Singh, M. Livny, G. B. Berriman, and J. Good. The cost of doing science on the cloud: The Montage example. In *Proc. High Performance Computing (SC'08)*, pages 1–12. IEEE/ACM, 2008.

[20] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by run-time parameter adaptation. In *Proc. ICSE'09*, pages 111–121. IEEE, 2009.

[21] E. Fersman, P. Krcál, P. Pettersson, and W. Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149–1172, 2007.

[22] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proc. POPL'09*, pages 127–139. ACM, 2009.

[23] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009.

[24] J. C. Jacob, D. S. Katz, G. B. Berriman, J. Good, A. C. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. A. Prince, and R. Williams. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *Intl. Journal of Computational Science and Engineering*, 4(2):73–87, 2009.

[25] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Proc. Symposium on Formal Methods for Components and Objects (FMCO)*, *LNCS* 6957, pages 142–164. Springer, 2011.

[26] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.

[27] E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Dynamic resource reallocation between deployment components. In *Proc. Intl. Conference on Formal Engineering Methods (ICFEM'10)*, *LNCS* 6447, pages 646–661. Springer, 2010.

[28] E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Validating timed models of deployment components with parametric concurrency. In *Formal Verification of Object-Oriented Software (FoVeOOS)*, *LNCS* 6528, pages 46–60. Springer, 2011.

[29] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Intl. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.

[30] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[31] A. Nuñez, J. Vázquez-Poletti, A. Caminero, G. Castañé, J. Carretero, and I. Llorente. iCanCloud: A flexible and scalable cloud infrastructure simulator. *Journal of Grid Computing*, 10:185–209, 2012.

[32] D. B. Petriu and C. M. Woodside. An intermediate metamodel with scenarios and resources for generating performance models from UML designs. *Software and System Modeling*, 6(2):163–184, 2007.

[33] M. Verhoef, P. G. Larsen, and J. Hooman. Modeling and validating distributed embedded real-time systems with VDM++. In *Proc. Formal Methods (FM'06)*, *LNCS* 4085, pages 147–162. Springer, 2006.

CHAPTER **8**

# Paper 4: Simulating Concurrent Behaviors with Worst-Case Cost Bounds *

***Authors:*** Elvira Albert, Samir Genaim, Miguel Gómez-Zamalloa, Einar Broch Johnsen, Rudolf Schlatte, Silvia Lizeth Tapia Tarifa

***Publication:*** Proceedings of the 17th International Symposium on Formal Methods, volume 6664 of Lecture Notes in Computer Science, pages 353–368. Springer, June 2011

***Abstract:*** Modern software systems are increasingly being developed for deployment on a range of architectures. For this purpose, it is interesting to capture aspects of low-level deployment concerns in high-level modeling languages. In this paper, an executable object-oriented modeling language is extended with resource-restricted deployment components. To analyze model behavior a formal methodology is proposed to assess resource consumption, which balances the scalability of the method and the reliability of the obtained results. The approach applies to a general notion of resource, including traditional cost measures (e.g., time, memory) as well as concurrency-related measures (e.g., requests to a server, spawned tasks). The main idea of our approach is to combine reliable (but expensive) worst-case cost analysis of statically predictable parts of the model with fast (but inherently incomplete) simulations of the concurrent aspects in order to avoid the state-space explosion. The approach is illustrated by the analysis of memory consumption.

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

## 8.1 Introduction

Software systems today are increasingly being developed to be highly configurable, not only with respect to the functionality provided by a specific instance of the system but also with respect to the targeted deployment architecture. An example of a development method is software product line engineering [20]. In order to capture and analyze the intended deployment variability of such software, formal models need to express and range over different *deployment scenarios*. For this purpose, it is interesting to reflect aspects of low-level deployment in high-level modeling languages. As our first contribution, in this paper, we propose a notion of *resource-restricted* deployment component for an executable modeling language based on *concurrent objects* [8, 11, 14, 21, 24]. The main idea of resource-restricted deployment components is that they are parametric in the amount of resources they make available to their concurrently executing objects. This way, different deployment scenarios can be conveniently expressed at the modeling level and a model may be analyzed for a range of deployment scenarios.

As our main contribution, we develop a novel approach for estimating the *resource consumption* of this kind of resource-constrained concurrent executions which is reasonably reliable and scalable. Resource consumption is in this sense a way of understanding and debugging the model of the deployment components. Our work is based on a general notion of *resource*, which can be any function that associates a cost unit to the program statements. Traditional resources are execution steps, time and memory, but one may also consider more concurrency-related resources like the number of tasks spawned along the execution, the number of requests to a server, etc.

The two main approaches to estimating resource consumption of a program execution are *static cost analysis* and *dynamic simulation* (or monitoring). Efficient simulation techniques can analyze model behavior in different deployment scenarios, but simulations are carried out for particular input data. Hence, they cannot guarantee the correctness of the model. Due to the non-determinism of concurrent execution and the choice of inputs, possible errors may go undetected in a simulation. Static cost (or resource usage) analysis aims at automatically inferring a program's resource consumption *statically*, i.e., without running the program. Such analysis must consider all possible execution paths and ensures soundness, i.e., it guarantees that the program never exceeds the inferred resource consumption for any input data. While cost analysis for sequential languages exists, the problem has not yet been studied in the concurrent setting, partly due to the inherent complexity of concurrency: the number of possible execution paths can be extremely large and the resulting outcome non-deterministic. Statically analyzing the concurrent behaviors of our resource-restricted models requires a full state space exploration and quickly becomes unrealistic.

In this paper, we propose to combine simulations with static techniques for cost

analysis, which allows classes of input values to be covered by a single simulation. The main idea is to apply cost analysis to the sequential computations while simulation handles the concurrent system behavior. Our method is developed for an abstract behavioral specification language *ABS*, simplifying Creol [11, 14], which contains a functional level where computations are sequential and an concurrent object-oriented level based on concurrent objects. This separation allows a concise and clean formalization of our technique. The combination of simulation and static analysis, as proposed in this paper, suggests a middle way between full state space exploration and simulating single paths, which gives interesting insights into the behavior of concurrent systems.

*Paper organization.* Section 8.2 describes the syntax and semantics of the ABS modeling language and introduces our running example. Section 8.3 discusses the worst-case cost bounds calculation of the functional parts of ABS. Section 8.4 introduces deployment components, which model resource-containing runtime entities, and Section 8.5 presents the results of applying the technique to the running example. Finally, Section 8.6 discusses related work and Section 8.7 concludes the paper with a discussion of extensions of our presented technique.

## 8.2 A Language for Distributed Concurrent Objects

Our method is presented for *ABS*, an abstract behavioral specification language for distributed concurrent objects (simplifying Creol [11, 14] by excluding, e.g., class inheritance and dynamic class upgrades). Characteristic features of ABS are that: (1) it allows abstracting from implementation details while remaining executable; i.e., a *functional sub-language* over abstract data types is used to specify internal, sequential computations; and (2) it provides *flexible concurrency and synchronization mechanisms* by means of asynchronous method calls, release points in method definitions, and cooperative scheduling of method activations.

Intuitively, concurrent ABS objects have dedicated processors and live in a distributed environment with asynchronous and unordered communication. All communication is between named objects, typed by interfaces, by means of asynchronous method calls. (There is no remote field access.) Calls are asynchronous as the caller may decide at runtime when to synchronize with the reply from a call. Method calls may be seen as triggers of concurrent activity, spawning new activities (so-called *processes*) in the called object. Active behavior, triggered by an optional *run* method, is interleaved with passive behavior, triggered by method calls. Thus, an object has a set of processes to be executed, which stem from method activations. Among these, at most one process is *active* and the others are *suspended* in a process pool. Pro-

| *Syntactic categories.* | *Definitions.* | | |
|---|---|---|---|
| $I$ in Interface type | $Dd$ | $::=$ | $\texttt{data}\ D = Cons;$ |
| $D$ in Data type | $Cons$ | $::=$ | $Co[(\overline{T})] \mid (Cons \mid Cons)$ |
| $x$ in Variable | $F$ | $::=$ | $\texttt{def}\ T\ fn(\overline{T\ x}) = e;$ |
| $e$ in Expression | $T$ | $::=$ | $I \mid D$ |
| $b$ in Bool Expression | $e$ | $::=$ | $b \mid x \mid t \mid \texttt{this} \mid Co[(\overline{e})] \mid fn(\overline{e}) \mid \texttt{case}\ e\ \{\overline{br}\}$ |
| $t$ in Ground Term | $t$ | $::=$ | $Co[(\overline{t})] \mid \texttt{null}$ |
| $br$ in Branch | $br$ | $::=$ | $p \Rightarrow e;$ |
| $p$ in Pattern | $p$ | $::=$ | $_ \mid x \mid t \mid Co[(\overline{p})]$ |

Figure 8.1: ABS syntax for the functional level. Terms $\overline{e}$ and $\overline{x}$ denote possibly empty lists over the corresponding syntactic categories, and square brackets [ ] optional elements. Boolean expressions $b$ include comparison by equality, greater- and less-than operators.

cess scheduling is non-deterministic, but controlled by *processor release points* in a cooperative way.

An ABS *model* defines interfaces, classes, datatypes, and functions, and has a `main` method to configure the initial state. Objects are dynamically created instances of classes; their declared attributes are initialized to arbitrary type-correct values, but may be redefined in an optional method *init*. This paper assumes that models are well-typed, so method binding is guaranteed to succeed.

*The functional level of ABS* defines data types and functions, as shown in Fig. 8.1. In data type declarations $Dd$, a data type $D$ has at least one constructor $Cons$, which has a name $Co$ and a list of types $T$ for its arguments. Function declarations $F$ consist of a return type $T$, a function name $fn$, a list of variable declarations $\overline{x}$ of types $\overline{T}$, and an expression $e$. *Expressions* $e$ include Boolean expressions $b$, variables $x$, (ground) terms $t$, the (read-only) variable **`this`** which refers to the object's identifier, constructor expressions $Co(\overline{e})$, function expressions $fn(\overline{e})$, and case expressions $\texttt{case}\ e\ \{\overline{br}\}$. Ground terms $t$ are constructors applied to ground terms $Co(\overline{t})$, and **`null`**. Case expressions have a list of branches $p \Rightarrow e$, where $p$ is a pattern. The branches are evaluated in the listed order. Patterns include wild cards `_`, variables $x$, terms $t$, and constructor patterns $Co(\overline{p})$. Remark that expressions may refer to object references.

**Example 18.** *Consider a polymorphic data type for sets and a function* `in` *which checks if* `e` *is an a member of the set* `ss`*.*

| *Syntactic categories.* | *Definitions.* | | |
|---|---|---|---|
| $C, m$ in Names | $IF$ | $::=$ | $\texttt{interface}\ I\ \{\ \overline{Sg}\ \}$ |
| $g$ in Guard | $CL$ | $::=$ | $\texttt{class}\ C\ [(\overline{T\ x})]\ [\texttt{implements}\ \overline{I}]\ \{\ \overline{T\ x};\ \overline{M}\}$ |
| $s$ in Statement | $Sg$ | $::=$ | $T\ m\ (\overline{T\ x})$ |
| | $M$ | $::=$ | $Sg\ \{\ \overline{T\ x};\ s\ \}$ |
| | $g$ | $::=$ | $b \mid x? \mid g \wedge g \mid g \vee g$ |
| | $s$ | $::=$ | $s; s \mid x := rhs \mid \texttt{release} \mid \texttt{await}\ g \mid \texttt{return}\ e$ |
| | | $\mid$ | $\texttt{if}\ b\ \texttt{then}\ \{\ s\ \}\ [\texttt{else}\ \{\ s\ \}] \mid \texttt{while}\ b\ \{\ s\ \} \mid \texttt{skip}$ |
| | $rhs$ | $::=$ | $e \mid \texttt{new}\ C[(\overline{e})] \mid [e]!m(\overline{e}) \mid x.\texttt{get}$ |

Figure 8.2: ABS syntax for the concurrent object level.

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

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

*The concurrent object level of ABS* is given in Fig. 8.2. Here, an interface $IF$ has a name $I$ and method signatures $Sg$. A class implements a list of interfaces, specifying types for its instances; a class $CL$ has a name $C$, interfaces $\overline{I}$, class parameters and state variables $x$ of type $T$, and methods $M$ (The *attributes* of the class are both its parameters and state variables). A method signature $Sg$ declares the return type $T$ of a method with name $m$ and formal parameters $\overline{x}$ of types $\overline{T}$. $M$ defines a method with signature $Sg$, a list of local variable declarations $\overline{x}$ of types $\overline{T}$, and a statement $s$. Statements may access attributes of the current class, locally defined variables, and the method's formal parameters.

Right hand side expressions $rhs$ include object creation **new** $C(\overline{e})$, method calls, and (pure) expressions $e$. Statements are standard for assignment $x := rhs$, sequential composition $s_1; s_2$, and **skip**, **if**, **while**, and **return** constructs. **release** unconditionally releases the processor, suspending the active process. In **await** $g$, the guard $g$ controls processor release and consists of Boolean conditions $b$ and return tests $x?$ (see below). If $g$ evaluates to false, the processor is released and the process *suspended*. When the processor is idle, any enabled process from the object's pool of suspended processes may be scheduled. Explicit signaling is therefore redundant. Like expressions $e$, guards $g$ are side-effect free.

*Communication* in ABS is based on asynchronous method calls, denoted $o!m(\overline{e})$.

(Local calls are written $!m(\overline{e})$.) After asynchronously calling $x := o!m(\overline{e})$, the caller may proceed with its execution without blocking on the call. Here $x$ is a future variable, $o$ is an object (an expression typed by an interface), and $\overline{e}$ are expressions. A future variable $x$ refers to a return value which has yet to be computed. There are two operations on future variables, which control external synchronization in ABS. First, a return test $x?$ evaluates to false unless the reply to the call can be retrieved. (Return tests are used in guards.) Second, the return value is retrieved by the expression $x$.**get**, which blocks all execution in the object until the return value is available. The statement sequence $x := o!m(\overline{e})$; $v := x$.**get** encodes a blocking, *synchronous call*, abbreviated $v := o.m(\overline{e})$, whereas the statement sequence $x := o!m(\overline{e})$; **await** $x?$; $v := x$.**get** encodes a non-blocking, *preemptable call*, abbreviated **await** $v := o.m(\overline{e})$.

**Example 19.** *Consider a model of a book shop where clients can order a list of `books` for delivery to a `country`. Clients connect to the shop by calling the `getSession` method of an `Agent` object. An `Agent` hands out `Session` objects from a dynamically growing pool. Clients call the `order` method of their `Session` instance, which calls the `getInfo` and `confirmOrder` methods of a `Database` object shared between the different sessions. `Session` objects return to the agent's pool after an order is completed. (The full model is available in [5].)*

```
interface Agent {
   Session getSession();
   Unit free(Session session);
}

interface Session {
    OrderResult order(List<Bname> books, Cname country);
}

interface Database {
    DatabaseInfo getInfo(List<Bname> books, Cname country);
    Bool confirmOrder(List<Bname> books);
}

class DatabaseImp(Map<Bname,Binfo> bDB, Map<Cname,Cinfo> cDB) implements Database {
    DatabaseInfo getInfo(List<Bname> books, Cname country){
        Map<Bname,Binfo> bOrder:=EmptyMap;
        Pair<Cname,Cinfo> cDestiny;
        bOrder:=getBooks(bDB, books);
        cDestiny:=getCountry(cDB, country);
        return Info(bOrder, cDestiny);
    }
    ...
```

*In the model, a `DatabaseImp` class stores and handles the information about the books available in the shop (in the `bDB` map) as well as information about the delivery countries (in the `cDB` map). This class has a method `getInfo`; given an order with a list of `books` and a destination `country`, the `getInfo` method extracts information about book availability from `bDB` and shipping information from `cDB` by means of function*

*calls* `getBooks(bDB, books)` *and* `getCountry(cDB, country)` *The result from the method call has type* `DatabaseInfo`*, with a constructor of the form:* `Info(bOrder, cDestiny)`*.*

### *8.2.1 Operational Semantics*

The operational semantics of ABS is presented as a transition system in an SOS style [19]. Rules apply to subsets of configurations (the standard context rules are not listed). For simplicity we assume that configurations can be reordered to match the left hand side of the rules (i.e., matching is modulo associativity and commutativity as in rewriting logic [18]). A run is a possibly nonterminating sequence of rule applications. When auxiliary functions are used in the semantics, these are evaluated in between the application of transition rules in a run.

*Configurations* $cn$ are sets of objects, invocation messages, and futures. The associative and commutative union operator on configurations is denoted by whitespace and the empty configuration by $\varepsilon$. These configurations live inside curly brackets; in the term $\{cn\}$, $cn$ captures the *entire* configuration. An *object* is a term $ob(o, C, a, p, q)$ where $o$ is the object's identifier and $C$ its class, $a$ an attribute mapping representing the object's fields, $p$ an *active process*, and $q$ a *pool of suspended processes*. A process $p$ consists of a mapping $l$ of local variable bindings and a list $s$ of statements, denoted by $\{l|s\}$ when convenient. In an *invocation message* $invoc(o, f, m, \overline{v})$, $o$ is the callee, $f$ the future to which the call's result is returned, $m$ the method name, and $\overline{v}$ the call's actual parameter values. A *future* $fut(f, v)$ has a identifier $f$ and a reply value $v$ (which is $\perp$ when the future's reply value has not been received). Values are object and future identifiers, Boolean expressions, and null (as well as expressions in the functional language). For simplicity, classes are not represented explicitly in the semantics, but may be seen as static tables.

*Evaluating Expressions.* Denote by $\sigma(x)$ the value bound to $x$ in a mapping $\sigma$ and by $\sigma_1 \circ \sigma_2$ the composition of mappings $\sigma_1$ and $\sigma_2$. Given a substitution $\sigma$ and a configuration $cn$, denote by $[\![e]\!]_\sigma^{cn}$ a confluent and terminating reduction system which reduces expressions $e$ to data values. Let $[\![x?]\!]_\sigma^{cn} = \text{true}$ if $[\![x]\!]_\sigma^{cn} = f$ and $fut(f, v) \in cn$ for some value $v \neq \perp$, otherwise $[\![x?]\!]_\sigma^{cn} = \text{false}$. The remaining cases are fairly straightforward, looking up values for declared variables in $\sigma$. For brevity, we omit the reduction system for the functional level of ABS (for details, see [5]) and simply denote by $[\![e]\!]_\sigma^{\varepsilon}$ the evaluation of a guard or expression $e$ in the context of a substitution $\sigma$ and a state configuration $cn$ (the state configuration is needed to evaluate future variables). The reduction of an expression always happens in the context of a given process, object state, and configuration. Thus, $\sigma = a \circ l$ (the composition of the fields $a$ and the local variable bindings $l$), and $cn$ the current configuration of the system (ignoring the object itself).

*Transition Rules.* Transition rules of the operational semantics transform state configurations into new configurations, and are given in Fig. 8.3. We assume given functions $bind(o, f, m, \overline{v}, C)$ which returns a process resulting from the method activation of $m$ in a class $C$ with actual parameters $\overline{v}$, callee $o$ and associated future $f$; $init(C)$ which returns a process initializing instances of class $C$; and $atts(C, \overline{v}, o, n)$ which returns the initial state of an instance of class $C$ with class parameters $\overline{v}$, identity $o$, and deployment component $n$. The predicate $fresh(n)$ asserts that a name $n$ is globally unique (where $n$ may be an identifier for an object or a future). Let *idle* denote any process $\{l|s\}$ where $s$ is an empty statement list. Finally, we define different assignment rules for side effect free expressions (*assign1* and *assign2*), object creation (*new-object*), method calls (*async-call*), and future dereferencing (*read-fut*). Rule *skip* consumes a **`skip`** in the active process. Here and in the sequel, the variable $s$ will match any (possibly empty) statement list. Rules *assign1* and *assign2* assign the value of expression $e$ to a variable $x$ in the local variables $l$ or in the fields $a$, respectively. (We omit the standard rules for if-then-else and while).

*Process Suspension and Activation.* Three operations are used to manipulate a process pool $q$: enqueue$(p, q)$ adds a process $p$ to $q$, $q \setminus p$ removes $p$ from $q$, and select$(q, a, cn, t)$ selects a process from $q$ (which is `idle` if $q$ is empty or no process is *ready* [14]). The actual definitions are left undefined; different definitions correspond to different process scheduling policies. Let $\emptyset$ denote the empty pool. Rule *release* suspends the active process to the pool, leaving the active process idle. Rule *await1* consumes the await statement if the guard evaluates to true in the current state of the object, rule *await2* adds a release statement in order to suspend the process if the guard evaluates to false. Rule *activate* selects a process from the pool for execution if this process is *ready* to execute, i.e., if it would not directly be resuspended or block the processor [14].

*Communication and Object Creation.* Rule *async-call* sends an invocation message to $o'$ with the unique identity $f$ (by the condition fresh$(f)$) of a new future, the method name $m$, and actual parameters $\overline{v}$. Note that the return value of the new future $f$ is undefined (i.e., $\bot$). Rule *bind-mtd* consumes an invocation method and places the process corresponding to the method activation in the process pool of the callee. Note that a reserved local variable 'destiny' is used to store the identity of the future associated with the call. Rule *return* places the return value into the call's associated future. Rule *read-fut* dereferences the future $f$ in the case where $v \neq \bot$. Note that if this attribute is $\bot$ the reduction in this object is *blocked*. Finally, *new-object* creates a new object with a unique identifier $o'$. The object's fields are given default values by atts$(B, \overline{v}, o', n)$, extended with the actual values $\overline{v}$ for the class parameters and $o'$ for this. In order to instantiate the remaining attributes, the process $p$ is loaded (we assume that this process reduces to idle if init$(B)$ is unspecified in the class definition, and that it asynchronously calls `run` if the latter is specified).

(SKIP)
$$\frac{}{ob(o,C,a,\{l|\text{skip};s\},q) \rightarrow ob(o,C,a,\{l|s\},q)}$$

(RELEASE)
$$\frac{}{ob(o,C,a,\{l|\text{release};s\},q) \rightarrow ob(o,C,a,\text{idle},\text{enqueue}(\{l|s\},q))}$$

(ASYNC-CALL)
$$\frac{o' = [\![e]\!]^{\varepsilon}_{(a\circ l)} \quad \overline{v} = [\![\overline{e}]\!]^{\varepsilon}_{(a\circ l)} \quad \text{fresh}(f)}{ob(o,C,a,\{l|x := e!m(\overline{e});s\},q) \rightarrow ob(o,C,a,\{l|x := f;s\},q)\ invoc(o',f,m,\overline{v})\ fut(f,\bot)}$$

(READ-FUT)
$$\frac{v \neq \bot \quad f = [\![e]\!]^{\varepsilon}_{(a\circ l)}}{ob(o,C,a,\{l|x := e.\text{get};s\},q)\ fut(f,v) \rightarrow ob(o,C,a,\{l|x := v;s\},q)\ fut(f,v)}$$

(RETURN)
$$\frac{v = [\![e]\!]^{\varepsilon}_{(a\circ l)} \quad l(\text{destiny}) = f}{ob(o,C,a,\{l|\text{return } e;s\},q)\ fut(f,\bot) \rightarrow ob(o,C,a,\{l|s\},q)\ fut(f,v)}$$

(NEW-OBJECT)
$$\frac{\text{fresh}(o') \ p = \text{init}(B) \quad a' = \text{atts}(B,[\![\overline{e}]\!]^{\varepsilon}_{a\circ l},o',n)}{ob(o,C,a,\{l|x := \text{new } B(\overline{e});s\},q) \rightarrow ob(o,C,a,\{l|x := o';s\},q)\ ob(o',B,a',p,\emptyset)}$$

(ASSIGN1)
$$\frac{x \in \text{dom}(l) \quad v = [\![e]\!]^{\varepsilon}_{(a\circ l)}}{ob(o,C,a,\{l|x := e;s\},q) \rightarrow ob(o,C,a,\{l[x \mapsto v]|s\},q)}$$

(AWAIT1)
$$\frac{\neg[\![g]\!]^{cn}_{(a\circ l)}}{\{ob(o,C,a,\{l|\text{await } g;s\},q)\ cn\} \rightarrow \{ob(o,C,a,\{l|\text{release};\text{await } g;s\},q)\ cn\}}$$

(ASSIGN2)
$$\frac{x \in \text{dom}(a) \quad v = [\![e]\!]^{\varepsilon}_{(a\circ l)}}{ob(o,C,a,\{l|x := e;s\},q) \rightarrow ob(o,C,a[x \mapsto v],\{l|s\},q)}$$

(AWAIT2)
$$\frac{[\![g]\!]^{cn}_{(a\circ l)}}{\{ob(o,C,a,\{l|\text{await } g;s\},q)\ cn\} \rightarrow \{ob(o,C,a,\{l|s\},q)\ cn\}}$$

(BIND-MTD)
$$\frac{p' = \text{bind}(o,f,m,\overline{v},C)}{ob(o,C,a,p,q)\ invoc(o,f,m,\overline{v}) \rightarrow ob(o,C,a,p,\text{enqueue}(p',q))}$$

(ACTIVATE)
$$\frac{p = \text{select}(q,a,cn)}{\{ob(o,C,a,\text{idle},q)\ cn\} \rightarrow \{ob(o,C,a,p,q\backslash p)\ cn\}}$$

Figure 8.3: ABS Semantics

## 8.3 Worst-Case Cost Bounds

The goal of this section is to infer *worst-case upper bounds* (UBs) from the (sequential) functions in our sub-language. This problem has been intensively studied since the seminal paper on cost analysis [23]. Thus, instead of a formal development, we illustrate the main steps of the analysis on the running example.

***Size of terms.*** The cost of a function that traverses a term $t$ usually depends on the *size* of $t$, and not on the concrete data structure to which $t$ is bound. For instance,

the cost of executing $dom(map)$ (which returns the domain of a map) depends on the size of $map$ (the number of elements). Therefore, in order to infer worst-case UBs, we first need to define the meaning of *size of a term*. This is done by using *norms* [7]. A norm is a function that maps terms to their size. For instance, the *term-size* norm calculates the number of type constructors in a given term, and is defined as $|Co(t_1,\ldots,t_n)|_{ts} = 1+\Sigma_{i=1}^{n}|t_i|_{ts}$, and, the *term-depth* norm calculates the depth of the term, and is defined as $|Co(t_1,\ldots,t_n)|_{td} = 1+\max(|t_1|_{td},\ldots,|t_n|_{td})$. Consider the book shop model described in Ex. 19; the database uses maps for storing information; a `Map<A, B>` has two constructors `Ins(Pair<A, B>, Map<A, B>)` and `EmptyMap` (to represent empty maps). For storing the information of a book sold by the shop, the model uses a constructor of the form `BInfo(Bquantity, Bweight, Bbackordertime)` (A more detailed description of this data type can be found in [5].). For a term:

$t =$ `Ins(Pair("b1",BInfo(5,1,2)),Ins(Pair("b2",BInfo(1,2,5)),EmptyMap))`

which can represent the database of books in the shop, we have that $|t|_{ts} = 15$ and $|t|_{td} = 5$. Note that we count strings and numbers as type constructors. Norms map a given variable $x$ to itself in order to account for the size of the term to which $x$ is bounded. Any norm can be used in the analysis, depending on the used data structures, w.l.o.g., we will use the term-size norm.

***Size relations.*** The `getBooks` function (called from method `getInfo` in Ex. 19) returns a sub-database (of `booksDB`) which contains only those books in `books`:

```
def Map getBooks(Map booksDB,List books) =
  case books {
    Nil => EmptyMap;
    Cons(b,t) =>
       case in(dom(booksDB),b) {
         False => getBooks(booksDB,t) ;
         True => Ins(Pair(b,lookup(booksDB,b)),getBooks(booksDB,t));
       };
   };
```

Function `dom` returns the set of keys of the mapping provided as argument, `in` is the one of Ex. 18, and, `lookup` returns the value that corresponds to the provided key in the provided mapping. Observe that the return value of `dom` is passed on to function `in`. Since the cost of `in` is part of the total cost of `getBooks`, we need to express its cost in terms of `booksDB`. This is possible only if we know which is the relation between the returned value of `dom` and its input value `booksDB`. This *input-output* relation (or a post-condition) is a conjunction of (linear) constraints that describe a relation between the sizes of the input parameters of the function and its return value, w.r.t. the selected norm. E.g., $ret \leq map$ is a possible post-condition for function $dom$, where $map$ is the size of its input parameter and $ret$ is the size of the returned term.

We apply existing techniques [6] to infer such relations for our functional language. In what follows, we assume that $\mathcal{I}_P$ includes a post-conditions $\langle fn(\bar{x}), \psi\rangle$ for each function, where $\psi$ is a conjunction of (linear) constraints over $\bar{x}$ and $ret$.

***Cost Model.*** Cost analysis is typically parametric on the notion of *cost model* $\mathcal{M}$, i.e., on the resource that we want to measure [2]. Informally, a cost model is a function that maps instructions to costs. Traditional cost models are: (1) *number of instructions*, which maps all instructions to 1, i.e., $\mathcal{M}(b) = 1$ for all instructions $b$; and (2) *memory consumption*, which can be defined as $\mathcal{M}_h(x = t) = \mathcal{M}_h(t) = mem(t)$ where $mem(Co(t_1, \dots, t_n)) = \mathsf{Co} + \Sigma_{i=1}^{n} mem(t_i)$ and $mem(x) = 0$. For any other instruction $b$ we let $\mathcal{M}_h(b) = 0$. The symbol $\mathsf{Co}$ represents the amount of memory required for constructing a term of type $Co$. Note that we estimate only the memory required for storing terms.

***Upper bounds.*** In order to make the presentation simpler, we assume functions are normalized such that nested expressions are flattened using **`let`** bindings. Using this normal form, the evaluation of an expression $e$ consists in evaluating a sequence of sub-expressions of the form $y = fn(\bar{x})$, $y = t$, $match(y, t)$, $fn(\bar{x})$, $t$ or $x$. We refer to such sequence as an execution path of $e$. In a static setting, since variables are not assigned concrete values, and due to the use of **`case`**, an expression $e$ might have several execution paths. We denote the set of all execution paths of $e$ by $paths(e)$. This set can be constructed from the abstract syntax tree of $e$. Clearly, when estimating the cost of executing an expression $e$ we must consider all possible execution paths. In practice, we generate a set of (recursive) equations where each equation accounts for the cost of one execution path. Then, the solver of [1] is used in order to obtained a closed-form UB.

**Definition 1.** *Given a function* **`def`** $T\ fn(\overline{T\ x}) = e$*, its cost relation (CR) is defined as follows: for each execution path* $p \equiv b_1, \dots, b_n \in paths(e)$*, we define an equation* $\langle fn(\bar{x}) = \Sigma_{i=1}^{n}\mathcal{M}(b_i) + fn_{i_1}(\bar{x}_{i_1}) + \cdots + fn_{i_k}(\bar{x}_{i_k}), \wedge_{i=1}^{n}\varphi_i\rangle$ *where* $fn_{i_1}(\bar{x}_{i_1}), \dots, fn_{i_k}(\bar{x}_{i_k})$ *are all function calls in* $p$*; and* $\varphi_i \equiv y = |t|_{ts}$ *if* $b_i \equiv y = t$*, and* $\varphi_i \equiv \psi[ret/y]$ *if* $b_i \equiv y = f(\bar{x})$ *and* $\langle f(\bar{x}), \psi\rangle \in \mathcal{I}_P$*, otherwise* $\varphi_i = true$*. The CR system of a given program the set of all CRs of its functions.*

**Example 20.** *The following is the CR of* `getBooks` *w.r.t the cost model mem:*

$$
\begin{array}{lcll}
getBooks(a,b) & = & \mathsf{EmptyMap} & \{b=1\} \\
getBooks(a,b) & = & dom(a){+}in(d,e){+}getBooks(a,g) & \{b=1{+}e{+}g, d{\le}a, d{\ge}1, e{\ge}1, g{\ge}1\} \\
getBooks(a,b) & = & \mathsf{Pair}{+}\mathsf{Ins}{+}dom(a){+}in(d,e) & \{b=1{+}e{+}g, d{\le}a, d{\ge}1, e{\ge}1, g{\ge}1\} \\
 & & +\ lookup(a,e){+}getBooks(a,g) &
\end{array}
$$

*The first equation can be read as "the memory consumption of getBooks is one EmptyMap constructor if the size of b is 1". Equations for functions in, lookup*

*and dom are not shown due to space limitations and have resp. constant, zero and linear memory consumptions. Solving the above CR results in the UB*

$$getBooks(a,b) = \mathsf{EmptyMap}+\mathsf{nat}(\tfrac{b-1}{2})*(\mathsf{nat}(\tfrac{a-1}{4})*\mathsf{Ins}+\mathsf{EmptySet}+\max(\mathsf{True},\mathsf{False}))$$

*Replacing, for example, EmptyMap, Ins, True and False by* 1 *results in*

$$getBooks(a,b) = 1 + \mathsf{nat}(\tfrac{b-1}{2}) * (2 + \mathsf{nat}(\tfrac{a-1}{4}))$$

# 8.4 Deployment Components

*Deployment components* make quantifiable deployment-level resources explicitly available in the modeling language. A deployment component allows the logical execution environment of concurrent objects to be mapped to a model of physical resources, by specifying an abstract execution context which is shared between a number of concurrently executing objects. The resources available to a deployment component are shared between the component's objects. An object may get and return resources from and to its deployment component. Thus, the deployment components impose a resource-restricted execution context for their concurrently executing objects, but not a communication topology as objects still communicate directly with each other independent of the components.

Resource-restricted deployment components are integrated in the modeling language as follows. Let variables $x$ of type `Component` refer to deployment components and allow deployment components to be statically created by the statement $x$:=component($r$) in the main method, which allocates a given quantity of resources $r$ to the component $x$ (capturing the resource constraint of $x$). Resources are modeled by a data type `Resource` which extends the natural numbers with an "unbounded resource" $\omega$. Resource allocation and usage is captured by resource addition and subtraction, where $\omega + n = \omega$ and $\omega - n = \omega$.

Concurrent objects residing on components, may grow dynamically. All objects are created inside a deployment component. The syntax for object creation is extended with an optional clause to specify the targeted deployment component in the expression **new** $C(\bar{e})$@ $x$. This expresses that the new $C$ object will reside in the component $x$. Objects generated without an @ clause reside in the same component as their parent object. Thus the behavior of an ABS model which does not statically declare additional deployment components can be captured by a root deployment component with $\omega$ resources.

**Example 21.** *Consider the book shop model described in Ex. 19 instantiated inside deployment components:*

| Rule | cost | free |
|---|---|---|
| ASSIGN1, ASSIGN2 | $\text{cost}(e)$ | $\|vp\| - \|v\|$ |
| READ-FUT | $max(\text{cost}(e), \|v\|)$ | 0 |
| BIND-MTD | $P + \|\overline{v}\|$ | $-(P + \|\overline{v}\|)$ |
| ASYNC-CALL | $\text{cost}(\overline{e}) + \|f\|$ | 0 |
| NEW-OBJECT-CREATE | $O + P + \|\overline{v}\|$ | $-(O + P + \|\overline{v}\|)$ |

Table 8.1: The non-trivial cost functions of memory-constrained ABS semantics. All identifiers are the same as in the corresponding rule of Figure 8.3, except $vp$ (old value of a variable), $|v|$ (size of term $v$), $P$ (size of a process), and $O$ (size of an object).

```
Component c := component(200);
Database db := new DataBaseImp(...) @ c;
Agent agent := new AgentImp(db) @ c;
```

*The `Session` objects created and handed out by the `Agent` object will then be created inside `c` as well, without further changes to the model.*

The *execution* inside a component $d$ with $r$ resources can be understood as follows. An object $o$ residing in $d$ may execute a transition step with cost $c$ if

- $o$ can execute the step in a context with unbounded resources, and
- $c \leq r$; i.e., the cost of executing the step does not exclude the transition in an execution context restricted to $r$ resources.

After the execution of the transition step, the object may return free resources to its deployment component. Thus, for each transition rule the resources needed to apply this rule to a state $t$, resulting in a state $t'$, can be characterized in terms of two functions over the state space, one computing the *cost* of the transition form $t$ to $t'$ and one computing the *free* resources after the transition. The allocation and return of resources for objects in a deployment component will depend on the specific cost model $\mathcal{M}$ for the considered resource, so the exact definitions of $cost_{\mathcal{M}}(t, t')$ and $free_{\mathcal{M}}(t, t')$ depend on $\mathcal{M}$.

**Example 22.** *Table 8.1 shows the $cost_{\mathcal{M}}(t, t')$ and $free_{\mathcal{M}}(t, t')$ functions for the memory cost model of the ABS semantics, using the symbols of Figure 8.3. There are some subtle details in these functions – for example, message invocations and future variables are considered to be outside any one deployment component, so the memory required to execute the* READ-FUT *rule can be larger than evaluating the future variable expression $e$ since the deployment component must have enough memory to accommodate the incoming value $v$. Also, object creation affects two places, so was split into two rules, similar to method invocation.*

$$\frac{\begin{array}{cc} & (\textsc{Context}) \\ mycomp(o) = id & cost_{\mathcal{M}}(o\ \overline{msg}, o'\ \overline{msg}'\ \overline{config}') \leq r \\ o\ \overline{msg} \longrightarrow o'\ \overline{msg}'\ \overline{config}' & r' = r + free_{\mathcal{M}}(o\ \overline{msg}, o'\ \overline{msg}'\ \overline{config}') \end{array}}{\{comp(id, r)\ o\ \overline{msg}\ \overline{config}\} \longrightarrow_{\mathcal{M}} \{comp(id, r')\ o'\ \overline{msg}'\ \overline{config}\ \overline{config}'\}}$$

Figure 8.4: An operational semantics for resource-constrained deployment components

***Semantics of Resource Constrained Execution.*** Let $\mathcal{M}$ be a cost model. The operational semantics of $\mathcal{M}$-constrained execution in deployment components is defined as a small-step operational semantics, extending the semantics of ABS given in Sec. 8.2.1 to resource-sensitive runtime configurations for $\mathcal{M}$. We assume given functions $cost_{\mathcal{M}}(t, t')$ and $free_{\mathcal{M}}(t, t')$.

Let $\longrightarrow$ denote the single-step reduction relation of the ABS semantics, defined in Sec. 8.2.1. A resource-constrained run of an ABS model consists of zero or more applications of a transition relation $\longrightarrow_{\mathcal{M}}$, which is defined by the context rule given in Fig. 8.4. Runtime configurations are extended with the representation of deployment components $comp(id, r)$, where $id$ is the identifier of the component and $r$ its currently available resources. Each object has a field `mycomp`, instantiated to its deployment component at creation time (we omit the redefined object creation rule). Let $\overline{config}$ denote a set of objects and futures. The context rule expresses how an object $o$ may evolve to $o'$ given a set of invocation messages $\overline{msg}$ in the context of a deployment component with $r$ available resources. Since $o$ may consume an invocation message and create new objects, futures, or invocation messages, the right hand side of the rule returns $o'$ with a possibly different set of messages $\overline{msg}'$ and a configuration $\overline{config}'$.

# 8.5 Simulation and Experimental Results

To validate the approach presented in this paper, an interpreter for the ABS language was augmented with a resource constraint model that simulates systems with limited memory. The semantics of this ABS interpreter is given in rewriting logic [18] and executes on the Maude platform [10]. Note that the semantics of Section 8.4, when implemented directly, leads to a significant amount of backtracking in an actual simulation. For this reason, our Maude interpreter was modified to incorporate deployment components and use the costs of Table 8.1 for the execution of statements. One such modified rule is shown in Figure 8.5: An assignment to $x$ can only proceed if the cost of evaluating the right-hand side $e$ of the assignment statement is less than the currently free memory $r$. In this case, $x$ is bound to the new value $v$, and $r$ is adjusted using Table 8.1 (here, the difference between $v$ and the previous value $vp$). All other transition rules which evaluate expressions are modified in the same way.

$$\frac{\overset{\text{(ASSIGN1-RSC)}}{x \in \mathrm{dom}(l) \quad v = [\![e]\!]^{\varepsilon}_{(a \circ l)} \quad vp = l(x) \quad \mathrm{cost}(e) \leq r \quad \mathrm{mycomp}(o) = dc}}{\begin{array}{c} dc(r)\ ob(o, C, a, \{l|x := e; s\}, q) \\ \rightarrow dc(r + |vp| - |v|)\ ob(o, C, a, \{l[x \mapsto v]|s\}, q) \end{array}}$$

Figure 8.5: Resource-aware assignment rule, with an object $ob$ and deployment component $dc$. The assignment statement is only executed if $e$ can be evaluated with the current $r$, which is adjusted afterwards.

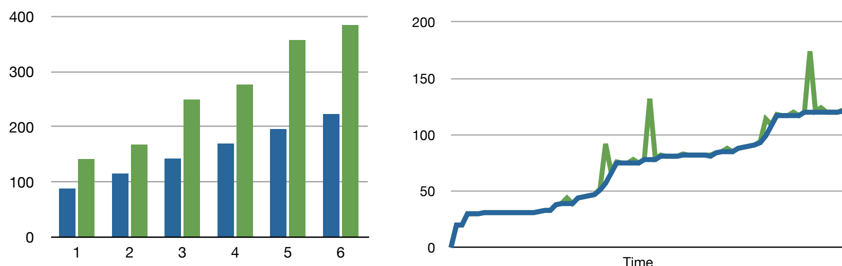


Figure 8.6: Final and peak memory use as a function of the size of input (left) and progression of memory use for execution using input size 2 (right).

*Simulation results.* Deployment component declarations were added to the book shop model described in Example 19, restricting the memory available to all objects of type `Database`, `Agent`, and `Session` (i.e., the server part of the model). Cost functions were computed for all functions in the model, as described in Sec. 8.3 (UBs for all functions in the book shop model can be found in [5]). With this interpreter, creating a deployment component with too little memory leads to the expected deadlock.

To obtain quantitative results, the interpreter was instrumented to record current memory $r$ and peak memory usage $r + \mathrm{cost}(s)$ during the evaluation of its resource-aware rules. This instrumentation yields both *maximum resource usage* and a *time series* of memory usage for a simulation run. Figure 8.6 (left) shows the peak intermediate memory usage and memory use at the end of the simulation for various input sizes (i.e., how often to run book orders of constant size). Figure 8.6 (right) shows the memory use over time of one single run of the model. The "peaks" in the right-hand side graph occur during evaluation of functions with large intermediate memory usage (the blue line represents memory use between execution steps, when the transient memory has been freed again).

## 8.6 Related Work

Static cost inference for sequential programming languages has recently received considerable attention. A cost analysis for Java bytecode has been developed in [2], for C++ in [12], and for functional programs in [13]. Our approach for inferring cost for the functional part of ABS is based on [2], which follows the classical approach of [23]. Inference of worst-case UBs on the memory usage of Java like programs with garbage collection is studied in [4]. The analysis accounts for memory freed by garbage collection, and thus infers more tight and realistic bounds. The analysis supports several GC schemes. The analysis of [13] supports inference of memory usage, and accounts for memory freed by destructive matching. In [16] live heap space analysis for a concurrent language has been proposed. However it uses a very limited model of shared memory. Recently, a cost analysis for X10 programs [9] has been developed [3], which infers UBs on the number of tasks that can be running in parallel. The concurrency primitives of X10 are similar to ABS, but X10 is not based on concurrent objects.

Formal resource modeling happens mainly in the embedded domain. For example, Verhoef et al. [22] use the timed VDM++ to model processing time, schedulability and bandwidth resources for distributed embedded systems, but their approach is less general and not used for memory consumption. Johnsen et al. modeled processing resources in the context of deployment components in previous work [15], but this work does not use cost analysis methods. There is not much work combining static cost analysis and simulation to analyze resource usage. However, Künzli et al. [17] combine exact simulation and arrival curves to model processing costs, decreasing the needed simulation time by using arrival curves in their simulations to abstract from some of the components in a SystemC model of specific hardware. In contrast, we use cost analysis to generalize simulations on abstract, formally defined object-oriented models.

## 8.7 Discussion

Software is increasingly being developed to be configured for different architectures, which may be restricted in the resources they provide to the software. Therefore, it is interesting to capture aspects of low-level deployment concerns at the abstraction level of a software modeling language. In this paper, we have shown how a formally defined executable concurrent object-oriented modeling language can be extended with a notion of deployment component, which imposes a resource-constraint on the execution of objects in the model.

In order to validate the behavior of the resource-restricted model, we propose to combine static cost analysis with simulations. This combination is achieved by ap-

plying static cost analysis to the sequential parts of the modeling language, for which practical cost analysis methods exist, while using simulation for the concurrent part, for which static approaches would lead to a state-space explosion. Thus, the complexity of applying static cost analysis to concurrent executions is avoided, and, in addition, we obtain better results than concrete simulations because the sequential parts of the model are simulated by the worst-case bounds. The technique is demonstrated for memory consumption analysis on an example. The analysis of memory consumption considered here could be strengthened by allowing explicit scheduling and garbage collection policies to be included in the model. This is left for future work.

Another interesting issue is how resource analysis carries over from executable models to generated code. A code generator from ABS to Java is under development that translates user defined abstract data types in ABS into object structures. Hence, the symbolic UBs inferred for memory consumption of the ABS models correspond to bounds on the number of objects in the corresponding Java code. Note that it might not be possible to find similar correlations for other cost models such as the number of execution steps. Another line of interesting future work is to set up actual measurements on generated code and use these results to profile our analysis approach for a given cost model.

# Bibliography

[1] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning* 42(6):161–203, 2011.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *Proc. ESOP'07*, LNCS 4421, pages 157–172. Springer, 2007.

[3] E. Albert, P. Arenas, S. Genaim, and D. Zanardini. Task-Level Analysis for a Language with Async-Finish parallelism. In *LCTES*. ACM Press, April 2011.

[4] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Parametric Inference of Memory Requirements for Garbage Collected Languages. In *ISMM*, ACM Press, 2010.

[5] E. Albert, S. Genaim, M. Gómez-Zamalloa, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Simulating concurrent behaviors with worst-case cost bounds. Research Report 403, Dept. of Informatics, Univ. of Oslo, Jan. 2011. `http://einarj.at.ifi.uio.no/Papers/rr403.pdf`

[6] F. Benoy and A. King. Inferring Argument Size Relationships with CLP(R). In *LOPSTR*, LNCS 1207, pages 204–223. Springer, 1997.

[7] A. Bossi, N. Cocco, and M. Fabris. Proving Termination of Logic Programs by Exploiting Term Properties. In *TAPSOFT*, LNCS 494.. Springer, 1991.

[8] D. Caromel and L. Henrio. *A Theory of Distributed Object.* Springer, 2005.

[9] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster computing. In *OOPSLA*, pages 519–538. ACM, 2005.

[10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude - A High-Performance Logical Framework*, LNCS 4350. Springer, 2007.

[11] F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *Proc. ESOP'07*, LNCS 4421, pages 316–330. Springer, 2007.

[12] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: Precise and Efficient Static Estimation of Program Computational Complexity. In *POPL*, ACM 2009.

[13] J Hoffmann, Klaus Aehlig, and M. Hofmann. Multivariate amortized resource analysis. In *POPL*, pages 357–370, ACM 2011.

[14] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, 2007.

[15] E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Dynamic resource reallocation between deployment components. In Proc. *ICFEM*, LNCS 6447, pages 646–661. Springer, 2010.

[16] M. Kero, P. Pietrzak, and N. J. Live Heap Space Bounds for Real-Time Systems. In *APLAS*, LNCS 6461, pages 287–303. Springer, 2010.

[17] S. Künzli, F. Poletti, L. Benini, and L. Thiele. Combining simulation and formal methods for system-level performance analysis. In *DATE*, pages 236–241. European Design and Automation Association, 2006.

[18] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[19] G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004.

[20] K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

[21] J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *Proc. ECOOP 2010*, LNCS 6183. Springer, 2010.

[22] M. Verhoef, P. G. Larsen, and J. Hooman. Modeling and validating distributed embedded real-time systems with VDM++. In *Proc. FM 2006*, LNCS 4085, pages 147–162. Springer, 2006.

[23] B. Wegbreit. Mechanical Program Analysis. *Comm. of the ACM*, 18(9), 1975.

[24] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proc. OOPSLA'05*, pages 439–453. ACM Press, 2005