Project N°: **FP7-231620**

Project Acronym: **HATS**

Project Title: **Highly Adaptable and Trustworthy Software using Formal Models**

Instrument: **Integrated Project**

Scheme: **Information & Communication Technologies**

Future and Emerging Technologies

Deliverable D2.1
A Configurable Deployment Architecture

Due date of deliverable: (T35)

Actual submission date: 1st February 2011

Start date of the project: **1st March 2009** Duration: **48 months**

Organisation name of lead contractor for this deliverable: **UIO**

Final version

Integrated Project supported by the 7th Framework Programme of the EC		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Executive Summary:
A Configurable Deployment Architecture

This document summarizes deliverable D2.1 of project FP7-231620 (HATS), an Integrated Project supported by the 7th Framework Programme of the EC within the FET (Future and Emerging Technologies) scheme. Full information on this project, including the contents of this deliverable, is available online at `http://www.hats-project.eu`.

The report develops the lowest levels of a flexible architecture upon which to base models in ABS, an abstract executable modeling language intended for modeling and analysis of software product families. This lowest level of ABS models captures deployment concerns, such as the underlying concurrency and interaction models as well as an abstract failure model for the language, are considered. The following aspects of ABS are reported:

Real-Time ABS is a timed extension of ABS which allows us to measure differences in performance depending on deployment choices;

Software Components structure ABS artefacts into logical units of composition, and provide isolation, mobility and data-flow reconfiguration capacities for distribution and dynamic redistribution of these units;

Deployment Components are used to model deployment choices such as CPU capacity, restricting the execution capacity of different parts of ABS models;

Abstract Failure Models explore the future variable mechanism of asynchronous calls in ABS to propose user-defined failures and compensations to both the server and client side of a method call; and

User-defined Scheduling allows the application developer to define different policies for giving priority to client requests.

List of Authors

Einar Broch Johnsen (UIO)
Rudolf Schlatte (UIO)
Mahdi Jaghoori (CWI)
Yannick Welsch (UKL)

Contents

Chapter 1

Introduction

The HATS deliverable **D2.1** *A configurable deployment architecture* reports on modeling deployment variability in ABS. Deployment variability is concerned with the lowest levels of a flexible architecture upon which to base ABS models. The HATS Description of Work (DoW) describes our intentions for modeling deployment choices and for extending the ABS modeling language with appropriate linguistic primitives. The description of work highlights explicit support for modeling aspects of the underlying deployment architecture such as concurrency, distribution, scheduling policies, and failure. We have focused on adding these features in a way which is incremental to the behavioral part of the models; i.e., deployment decisions can be added at any stage in the development of the model. In particular, these decisions can to a large extent be externalized into separate features. The specific integration of deployment architectures into the feature selection mechanisms of software product lines was discussed in deliverable **D1.2** *Full ABS Modeling Framework* [22] and will not be discussed in detail in this report. Here, we focus primarily on the integration of deployment architecture in the Core ABS modeling language [16, 31].

In developing modeling support for deployment architectures in ABS, a primary concern has been to lift low-level deployment aspects to the abstraction level of the modeling language; i.e., deployment decisions should be expressed in the same style as other aspects of a model. This is desirable, since in our experience using separate models or even different languages to describe different aspects of a system causes "model drift" and loss of coherence of the modeling effort as the individual models are changed in isolation. Thus, linguistic primitives for deployment configuration are tightly integrated into ABS. However, in order to support a separation of concerns between the behavioral part of the models and the deployment configurations, we take an incremental approach in which deployment decisions to a large extent are integrated into the models by means of optional annotations extending default deployment decisions. Many deployment aspects influence the quality-of-service level of a model rather than its functional behavior. For this reason, we first develop a timed version of ABS which allows us to measure performance of a model given a deployment architecture.

This deliverable reports on our work on the following aspects of deployment modeling:

Real-Time ABS integrates executable object-oriented modeling in ABS with linguistic primitives to specify real-time behavior. In particular, method invocations will have associated deadlines and the duality between blocking and suspending method activations in ABS is extended to include time constraints. Method activations have a local reference to their own deadlines, and may pass on parts of a deadline in an auxiliary call, or adapt their activity depending on the remaining available time. The approach is particularly suited for soft real-time requirements, such as the number of missed deadlines for a server in a given client scenario. This is particularly well-suited for measuring the consequences of deployment decisions in our context. Real-Time ABS is described in Chapter 2.

Software Components structure ABS artefacts into logical units of composition. We study a language COMP to describe the software architecture of ABS systems and their evolution at a high level of abstraction. COMP complements the existing ABS modeling approach by putting emphasis on the high-level hierarchical

structure of components, the capacity to move, update, and wrap components, and isolation capacities to encode distribution and wrapping. Chapter 3 describes COMP.

Deployment Components are proposed as a modeling concept to capture resource-restricted execution contexts for ABS artefacts. A deployment component controls the execution potential for a set of cogs (concurrent object groups) in ABS. Deployment components have been used to restrict execution with respect to both CPU capacity and memory constraints. In this report we mainly focus on CPU (work on memory is included in the Appendix). Deployment components are dynamically created artefacts in ABS, similar to objects. They may be created anywhere in an ABS model, but static deployment scenarios are typically defined in the main block of the model. Resources are also first-class citizens of ABS, which captures virtualized resources and allows ABS models to specify, for example, load-balancing between different deployment components. Thus, our approach covers not only static variability in the deployment modeling, but may also be used to capture dynamic deployment variability. Deployment components are described in Chapter 4.

Abstract Failure Models extend the use of futures in ABS to propagate failures. Futures are used in Core ABS to store reply values to asynchronous method calls. In the context of concurrent objects in ABS, we introduce a failure model inspired from web-services, in which the modeller can introduce abstract failures at both the client and server side of a method call. The future is perceived as a two-way communication channel related to the specific method call, used to communicate the occurrence of failures (in addition to regular replies to method calls). Our approach can be used to define failures at the server side (e.g., memory exhaustion) but also to kill a request from the client side. The approach deals with abstract failures and their corresponding compensations. Chapter 5 deals with failures.

User-defined Scheduling. We extend Real-Time ABS such that concurrent objects are parametric in their scheduling policy for pending method activations. In Core ABS, this scheduling policy is underspecified. We let an arbitrary scheduling policy be the default, and use optional annotations to introduce more fine-grained control of the scheduling. The schedulers themselves are modelled directly at the abstraction level of ABS by introducing processes as an algebraic data type in the functional part of the modeling language. Any user-defined scheduling function can be used in scheduling annotations, and applied to the scheduling by means of reflection in the operational semantics of Real-Time ABS. Chapter 6 describes user-defined schedulers at the modeling and simulation level, while Chapter 7 explores schedulability analysis for ABS.

1.1 List of Papers Comprising Deliverable D2.1

This section lists all the papers that comprise this deliverable, indicates where they were published, and explains how each paper is related to the main text of this deliverable. As requested by the reviewers, the papers are not directly attached to Deliverable D2.1, but are made available on the HATS web site at the following url: `http://www.hats-project.eu/sites/default/files/D2.1`. Direct links are also provided for each paper listed below.

Paper 1: Lightweight Time Modeling in Timed Creol

This paper [12] introduces an implicit time model for Creol, a predecessor of ABS based on concurrent objects with asynchronous method calls (but without concurrent object groups). The time model developed here is the first step in the direction of Real-Time ABS, as presented in Chapter 2.

This paper was written by Joakim Bjørk, Einar Broch Johnsen, Olaf Owe, and Rudolf Schlatte, and was published in the proceedings of RTRTS 2010.

Download Paper 1.

Paper 2: Dating Concurrent Objects: Real-Time Modeling and Schedulability Analysis

This paper [20] formalizes a model of real-time concurrent objects and shows how time-dependent scheduling of concurrent objects may be analyzed based on a timed automata framework. An intuitive understanding of the modeling concepts in this paper can be found in Chapter 2 and the analysis is explained in Chapter 7. The modeling concepts of this paper complement [12] in defining an explicit time model for concurrent objects, and represent the second step in the direction of Real-Time ABS, as presented in Chapter 2.

This paper was written by Frank S. de Boer, Mohammad Mahdi Jaghoori, and Einar Broch Johnsen, and was presented as an invited talk at CONCUR 2010.

Download Paper 2.

Paper 3: User-defined Schedulers for Real-Time Concurrent Objects

This paper [11] integrates the work on timed concurrent objects from [12, 20] with the additional artefacts of ABS to define Real-Time ABS. Real-Time ABS is introduced in Chapter 2. The paper then defines user-defined schedulers for Real-Time ABS in terms of default-overriding annotations. The intuitions behind this work are explained in Chapter 6.

This paper was written by Joakim Bjørk, Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa, and has been submitted for journal publication.

Download Paper 3.

Paper 4: A Component Model for the ABS Language

This paper [39] reports on initial work on the ABS component model COMP, including the hierarchical structure of components; the capacity to move, update, and wrap components; and isolation capacities to encode distribution and wrapping. This work is presented in Chapter 3.

This paper was written by Michael Lienhardt, Ivan Lanese, Mario Bravetti, Davide Sangiorgi, Gianluigi Zavattaro, Yannick Welsch, Jan Schäfer, and Arnd Poetzsch-Heffter. It was published in the proceedings of FMCO 2010.

Download Paper 4.

Paper 5: Validating Timed Models of Deployment Components with Parametric Concurrency

This paper [34] introduces the idea of deployment components as a modeling concept. In this paper, restrictions on CPU capacity were considered for a static deployment scenario which was defined in the main block of the ABS model. The processing resources were, however, given as a parameter to the deployment components, which makes it easy to compare performance of a model with different resource assumptions. In this paper, resource consumption is fixed by the ABS interpreter, which makes the approach quite lightweight for the modeller (compared to the explicit annotations in [35]). The intuitions behind this approach are explained in Chapter 4.

This paper was written by Einar Broch Johnsen, Olaf Owe, Rudolf Schlatte, and S. Lizeth Tapia Tarifa, and was published in the proceedings of FoVeOOS 2010.

Download Paper 5.

Paper 6: Dynamic Resource Reallocation Between Deployment Components

This paper [33] extends the linguistic primitives introduced in [34] to explicitly model resources as first-class citizens of ABS, enabling an ABS model to inspect the load over time in its deployment components and to specify load-balancing in dynamic deployment scenarios by transferring virtual resources between deployment components. The paper uses the same model for resource consumption as [34], and is discussed in Chapter 4.

This paper was written by Einar Broch Johnsen, Olaf Owe, Rudolf Schlatte, and S. Lizeth Tapia Tarifa, and was published in the proceedings of ICFEM 2010.

Download Paper 6.

Paper 7: A Formal Model of Object Mobility in Resource-Restricted Deployment Scenarios

This paper [36] complements [33] by considering load balancing based on object mobility rather than resource transfer. By combining deployment components with user-defined schedulers as introduced in Chapter 6, a pseudo-random scheduler was introduced to enable Monte-Carlo simulations for ABS models in different deployment scenarios. This is part of Chapter 4.

This paper was written by Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa and will be published in the proceedings of FACS 2011.

Download Paper 7.

Paper 8: Simulating Concurrent Behaviors with Worst-Case Cost Bounds

This paper [4] applies the idea of deployment components to memory restrictions, and combines static memory analysis using the COSTA tool for the functional part of ABS with simulations for the concurrent part. Deployment components with restricted memory extends the introduction of deployment components given in Chapter 4.

This paper was written by Elvira Albert, Samir Genaim, Miguel Gómez-Zamalloa, Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa, and was published in the proceedings of FM 2011.

Download Paper 8.

Paper 9: Fault in the Future

This paper [32] extends ABS with abstract failure models which allows the modeller to define failures. The paper takes an approach inspired from web services to define failure handling for concurrent objects, and defines a type system which ensures that all failures are handled. The intuitions behind this paper are explained in Chapter 5.

This paper was written by Einar Broch Johnsen, Ivan Lanese, and Gianluigi Zavattaro, and was published in the proceedings of COORDINATION 2011.

Download Paper 9.

Paper 10: Programming and Deployment of Active Objects with Application-Level Scheduling

This paper [45] shows how concurrent active objects (as in ABS) can be used for high-level scheduling of resources. A concurrent object provides a natural basis for a deployment scheme where each object virtually possesses one processor. The paper further proposes a tool architecture using Java to deploy applications based on this paradigm. This work is part of Chapter 6.

This paper was written by Behrooz Nobakht, Frank S. de Boer, Mohammad Mahdi Jaghoori, and Rudolf Schlatte. It will be published in the proceedings of SAC 2012.

Download Paper 10.

Paper 11: Networks of Real-Time Actors: Schedulability Analysis and Coordination

This paper [42] studies extending our general timed automata based framework for schedulability analysis to a coordination language to enable exogenous coordination of the objects. This extension provides a separation of concerns between computation and coordination. This work is presented in Chapter 7.

This paper was written by Mahdi Jaghoori, Ólafur Hlynsson, and Marjan Sirjani, and will be published in the proceedings of FACS 2011.

Download Paper 11.

Paper 12: From Nonpreemptive to Preemptive Scheduling: From Single-processor to Multiprocessor

This paper [28] studies decidable settings of non-preemptive to preemptive schedulers on single and multiple processor systems while tasks are specified themselves as timed automata. The paper first extends cooperative scheduling as used in ABS to preemptive scheduling with minimum inter-preemption delays and then the framework for schedulability analysis is extended to a multiprocessor setting and such that schedulability remains decidable. This work is presented in Chapter 7.

This paper was written by Mahdi Jaghoori and was published in the proceedings of SAC 2011.

Download Paper 12.

Chapter 2

Real-Time ABS

2.1 Introduction

One focus of Work Task T2.1 is on modeling deployment under resource-restricted scenarios, and how these scenarios influence timing properties of executable ABS models. To model and reason about time-sensitive models, the ABS language needs to incorporate a notion of time. This chapter summarizes the language extensions for ABS that were designed to achieve this goal.

UIO has developed a timed version of the ABS semantics in rewriting logic and extended the ABS interpreter in Maude, starting with work on real-time Creol objects by UIO [12] and by UIO and CWI [20]. The ABS language extension is described in [11] and summarized in this chapter. Furthermore, the expression `now()`, which was introduced in [12] (but not discussed in [11, 20]) is retained in our implementation and explained in Section 2.2.2, due to its convenience for logging purposes.

2.2 New Language Features

Timed executions allow the simulation of timed behavior of ABS models. This section presents the added language features that are used to specify timed behavior: method call deadlines, delays in execution and a notion of time tied to a global clock.

2.2.1 Concepts

Time is modeled by a global clock. Time advances (that is, the value of evaluating `now()` increases), when the following hold:

- No process in the running model is eligible for execution, all processes either wait for the result of some method invocation, or wait for time to advance.
- All method invocation messages have arrived at their target objects.

These two conditions mean that in each time interval the model executes "as much as possible" before the clock advances (run-to-completion semantics). In particular, code such as `while(True) { skip; }` will delay time advancement indefinitely. This semantics was chosen so that the developer can explicitly control delays in the model, for example before sending a message.[1]

A formal specification of these time advancement conditions can be found in [11] and [20].

[1] Section 4 shows how to combine notions of time and computation costs into a unified model – with computation cost, time-unaware models start exhibiting timed behavior.

2.2.2 Functional Level

Getting the current time The expression `now()` returns the value of the global clock.

Getting the current deadline The expression `deadline()` returns the current deadline, either as a term `Duration(`x`)` with decreasing $x \geq 0$, or as `InfDeadline` if no deadline was given to the method invocation.

Note that while these two expressions belong to the functional level since they are "pure" (do not modify the system state) and can be used inside arbitrary expressions, they are not functions in the mathematical sense since their value changes over time.

2.2.3 Concurrent Object Level

All language constructs in this section express relative time constraints.

Suspending a process The statement `await duration(min, max);` suspends the process for a time interval between *min* and *max*. Note that the process becomes schedulable after that time but is subject to normal scheduling policies, i.e., it is not guaranteed to run immediately the interval passes.

`await duration` is mostly used to model interactions with external systems, e.g., a file or database operation that is specified to take a specific amount of time but not modelled in detail.

Blocking a cog The statement `duration(min, max);` causes the current process to stay active and running for a time interval between *min* and *max*. During that time, no other process will become active in that cog. In contrast to `await duration`, the process is guaranteed to resume execution after at most *max* time units.

The `duration` statement is mostly used to model the time needed for execution in the current process.

Specifying process deadlines The statement `[Deadline: d]o!m();` creates a process in object `o` running the method `m`, with deadline `d`. The result of evaluating `deadline()` in the new process will be a duration x with $0 \leq x \leq d$.

2.3 Example

Here is a very simple, yet complete timed ABS model:

```
1  module DeadlineExample;
2
3  interface I { Bool m(); }
4  class C implements I {
5      Bool m() {
6          await duration(5, 5);
7          return durationValue(deadline()) > 0;
8      }
9  }
10
11 {
12     I i = new C();
13     [Deadline: Duration(6)] Bool call1 = i.m(); //returns True
14     [Deadline: Duration(4)] Bool call2 = i.m(); //returns False
15     Time t = now();
16 }
```

The method `m` of class `C` suspends for 5 time units, then returns `True` if the method deadline has been met, i.e. the deadline has not been decremented to 0 via clock advancements. The main block calls the method two times, with deadlines 4 and 6. One of the calls succeeds within the deadline, the other will exceed the deadline. At the end, the current time (`Time(10)` in this case) is recorded in the variable `t`.

Chapter 3

Software Components

3.1 Introduction

As part of Task 3.1 (Evolvable Systems), BOL and UKL studied software components and more specifically component evolution in the setting of HATS. We continue this development here. The modeling features that have been so far developed as part of HATS include classes, interfaces, deltas etc. However, these modeling features fail to describe systems at a high level of abstraction, as they miss the following features:

- a hierarchical structure of components,
- the capacity to move, update, and wrap components, and
- isolation capacities to encode distribution and wrapping.

The component model COMP we studied addresses these issues and complements the existing ABS modeling approach by providing a more architectural view on ABS systems and their evolution.[1] This is validated by encoding a number of important reconfiguration patterns using the primitives of COMP. In this chapter, we report on our FMCO 2010 paper [39] that presents a process calculus approach for component models that merges aspects of object-orientation and evolution.

Related Work and Goals

Many different component models have been developed over the past decade, like OSGi [5], Fractal [13], COM [17], Java Beans [48] and others [10, 41, 44]. These models focus on aspects different from those of COMP. If at all possible, dynamic evolution steps have to be hand-coded in these models (e.g., in OSGi, components can be stopped and replaced by new versions, but this has to be programmed using the framework API). In particular, high-level formal analysis of such steps is not supported. The central aspects and goals of COMP are summarized in the following.

Firstly, these previous models [5, 13, 17, 48] are not developed for formal analysis and do not provide a formal semantics needed for verification. In particular, they are more or less coupled to complex programming languages and APIs written in these languages, which makes it difficult to identify an analyzable formal core.

The second aspect is about interface specifications based on typing and ports. Many component models (e.g., [13, 40, 44]) support typed input and output ports to allow static checking of composition. To focus on dynamic evolution and keep the calculus manageable, we do not consider typing aspects in the core calculus presented in this paper. Similar to objects, a component has an interface consisting of a set of (untyped) methods. The component can communicate via channels with other components. In particular, the basic constructs of our core calculus can be used to model more structured connections between components. An extension to static analysis techniques (e.g., type systems) is planned as future work.

[1] The component model is not integrated at the language level of ABS, but rather complements it.

The third aspect concerns scope and component visibility. In many component models [13, 43, 47], the hierarchical structure of a component system is rigid: the boundary of a component a hides all the inner components, which are unreachable from a's environment. Other component models do not support nesting of components at all. COMP provides component nesting and flexibility with respect to visibility of internal components in order to model common patterns of distributed systems that involve sharing of resources.

The last issue is about the *passivation* mechanism provided by some of the models such as the Kell-calculus [47] or MECo [43]. Passivation allows the programmer to freeze a component and capture it; the component can then be sent around at will, with even the possibility of duplicating it. Passivation is powerful, but makes it quite hard to ensure safety of reconfiguration and to prove properties of systems. Also, the practical relevance of the act of copying a running component is doubtful.

3.2 The Component Model

To integrate well with ABS (and similar object-oriented languages), we adopt its basic scheme to handle components. As our main focus is on dynamic aspects, components here are runtime entities, often called component instances to distinguish them from programs. Components are represented by objects. The object's methods enable communication between the object and its environment. Furthermore, the same language specification technique is used to foster the integration between the calculus and the formal analysis techniques developed for ABS and Creol [3, 19].

COMP has a formal semantics, defined using the reduction and labeled transition system styles. Components in COMP have an input interface, but no output interfaces; as a consequence, components can be used much in the same way as objects; however, components additionally have mobility capacities. COMP provides opening and closing operations for dynamically changing the visibility of a component. Thus, while by default communication in COMP remains global to fit the communication capacities of objects, if needed, the boundary of a component a can be closed to restrict access to specific components internal to a, say b; as a consequence, a component external to a will not be able to directly access b any more. Mobility of running components is allowed by means of movement primitives rather than by capturing and communicating components. These primitives are inspired by the constructs for achieving mobility in the Ambient calculus [14]. In COMP a component may thus move in the tree hierarchy of a component system. Processes (including component definitions) may be communicated, but they cannot be grabbed when running. For a more detailed description of COMP, we refer to [39].

3.3 Example

To illustrate our component model, we use the ABS model of the Trading System Case Study [21]. We give the hierarchical partitioning of the Trading System in Figure 3.1. We distinguish between components that are part of the system to be developed (those within the Model box) and external entities (e.g. BarCode-ScannerEnv). The top-level components of the system are the cash desk lines, which are comprised of an express coordinator and multiple cash desks. Cash desks consist of different devices (e.g., bar code scanner, credit card reader, cash box, etc.). Many of these devices interact with the surrounding world, e.g., the bar code scanner scans the barcode of product items.

Isolation. A consequence of the component hierarchy is that a component may decide to hide its internal components, or to make (some of) them available to the external world. Hiding is fundamental for encapsulation: hidden components cannot be reached directly from the outside. Isolation can also be used to encode wrapping, where the wrapper component hides the wrapped one, while providing updated functionalities. In this way, for instance, methods can be removed, added or redefined. An internal component may however be left visible, which is useful for modeling shared resources. For example, the BarCodeScanner component can be used by the outside world in order to trigger product item scanning. Similarly, the CashDeskPC

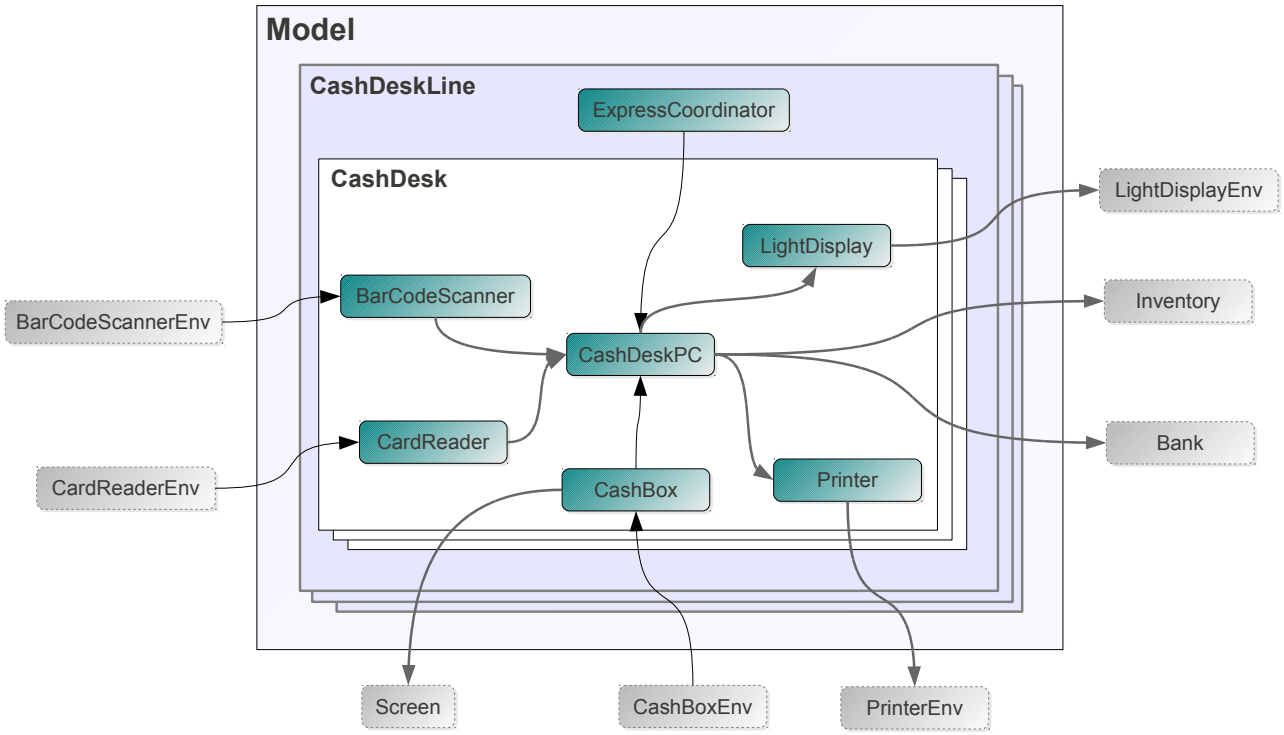

Figure 3.1: Component model of the Trading System Case Study

component is exposed to the surrounding CashDeskLine component so that it can triggered by the ExpressCoordinator component. On the other hand, the Printer component is not accessible from the outside world.

Mobility. Having a hierarchical structure in place, the immediate way for achieving evolution is to allow components to move along the hierarchy. Remember that mobility can be either logical or physical, according to whether components model the software architecture of the system or its physical distribution.

Clearly, different primitives for mobility are possible. We decided to introduce two primitives for mobility, `in` and `out`, inspired from the Ambient calculus [14], that allow us to move a component inside a sibling one or outside its parent. These primitives disallow direct mobility between locations far from each other (unrealistic in many cases, e.g., for physical locations). A typical example of mobility would be to move a CashDesk component to another CashDeskLine, i.e., outside of a specific CashDeskLine component and into another CashDeskLine component.

Chapter 4

Deployment Components

4.1 Introduction

In order to reflect deployment choices in a model, it is interesting to lift aspects of low-level deployment concerns to the abstraction level of the modeling language. UIO has extended ABS with a notion of *deployment components* in order to abstractly capture the consequences of resource restrictions related to deployment decisions at the modeling level. A deployment component is parametric in the amount of resources it provides to the ABS artefacts which are deployed on the component. This results in a flexible framework for variability of resources, which can be exploited in software product lines (see Deliverable D1.2 [22] for a discussion of this integration into delta modeling in ABS). The parameterization of resource capacity allows us to observe, by means of simulations, the performance characteristics of a system model under varying amounts of resources allocated to its parts.

This way, the modeler gains insight into the resource requirements of a component, in order to, for example, provide a minimum response time for given client behaviors. In the approach described in [33, 34], the resource consumption of the model was fixed by the ABS simulator or derived by the COSTABS tool [4], so the only parameter which could be controlled by the modeler was the capacity of the deployment component. In [35], we presented a way for the modeler to explicitly associate resource costs to different activities in a model. Resource costs may be specified by means of user-defined expressions in ABS; e.g., depending on the size of the actual input parameters to a method in the ABS model. Given a model with explicit cost specifications, the Maude back-end for ABS may be used for abstract performance analysis of formal object-oriented models, to analyze the performance of the model depending on the resources allocated to the deployment components. The approach allows simulations of performance at an earlier stage in the system design, by further abstraction from the control flow and data structures of the system under development.

We have explored a number of modeling possibilities within this framework of deployment components, to determine the best way to integrate the approach into the ABS language and to balance ease of use with precision and flexibility in practical modeling:

- UIO has defined a version in which resource management was hard-coded into the language interpreter for CPU resources, such that resource management was fixed but without overhead for the modeller, presented at FOVEOOS 2010 [34];
- UIO has defined a version in which (virtual) CPU resources may be dynamically reallocated at the application-level, based on for example the load on the deployment components, reported at ICFEM 2010 [33];
- UIO and UPM have shown how COSTA may be used to derive resource management for memory resources, with little overhead for the modeller but also with less possibilities for abstract models (e.g., resource-intensive skip statements cannot be inferred by COSTA), reported at FM 2011 [4]; and
- UIO has defined a version of concurrent object groups in ABS that may be dynamically moved between

deployment components. Object movement is controlled at the application-level based on, for example, the load on the deployment components. This work was presented at FACS 2011 [36].

In contrast to some of the published work (e.g., [35]), which presents a generic cost model $\mathcal{M}$ and introduces an explicit statement for resource consumption in ABS, in the HATS tool chain we have opted for a layered system, in which automatically defined costs are used by default but may be overridden by the modeller by means of annotations to introduce more fine grained resource management where desirable. In the ABS tool chain, we currently focus on CPU resources. Also existing in prototype form, but not in the mainline ABS tools, are different extensions for resource balancing as mentioned above and reported in the papers [33, 36].

4.2 New Language Features

All identifiers and datatypes described in this section are contained in the module `ABS.DC`. Models wishing to use deployment components must import the appropriate names or use fully-qualified names since this module is not imported by default.

The new language features extend, in a way, the work on Real-Time ABS (reported in Chapter 2), by adding implicit time advance that is caused by run-time costs incurred during simulation of models. In contrast to "pure" Real-Time ABS, by using deployment components simulated time advances not only through explicit `duration` and `await duration` statements, but also by executing normal statements which carry a certain *cost* of execution.

4.2.1 Concepts

The basic concept of our approach is a *deployment component*, on which concurrent object groups are deployed. By analogy with the software components of a software architecture, the deployment components of a model defines its deployment architecture; i.e., the deployment components reflect at the modeling level the machines on which the objects execute. Deployment components act as resource-restricted execution contexts for concurrent object groups in ABS: the resources which are available on the component control the amount of activity which can take place between two observable points in time. Thus, the work on deployment components extends Real-Time ABS as reported in Chapter 2. Deployment components are dynamically created (similar to objects) and provide *parametric concurrency* to the ABS models; i.e., the amount of concurrent processing capacity for the objects is determined by the CPU resources provided to their deployment component. This means that it is straightforward to vary the capacity of a component in order to adapt a product in a software product line in ABS to a specific deployment.

We can think of the deployment components of a model as its deployment architecture, in which the amount of resources allocated to the different components may vary. When we fix the amounts of resources for a deployment architecture, we get a specific *deployment scenario*. The resources given to a deployment component abstractly capture the capacity of the component. This means that execution in ABS objects must use this capacity, which is typically expressed in terms of a *cost model* for the resource, i.e., a relation from statements to costs. However, in order to introduce abstraction (compared to a programming language), we enable the modeller to associate specific costs with the specific statements of the model. For example, the modeller might want to abstract away an expensive computation by using a `skip` statement with a high associated cost. Therefore, we have opted for a system of defaults and optional annotations to express resource costs in a flexible yet lightweight way in ABS.

Given deployment scenarios for an ABS model, we can use the Real-Time ABS interpreter in Maude to perform experiments with the model. For this purpose, we can create a test case in which we let client objects interact with the deployed model, and measure, e.g., response times or deadline misses. The typical scenario is depicted in Figure 4.1.

In this chapter, we focus on the extension of ABS to express deployment variability "in space": we discuss linguistic extensions for static deployment scenarios (i.e., without object mobility), focusing on processing

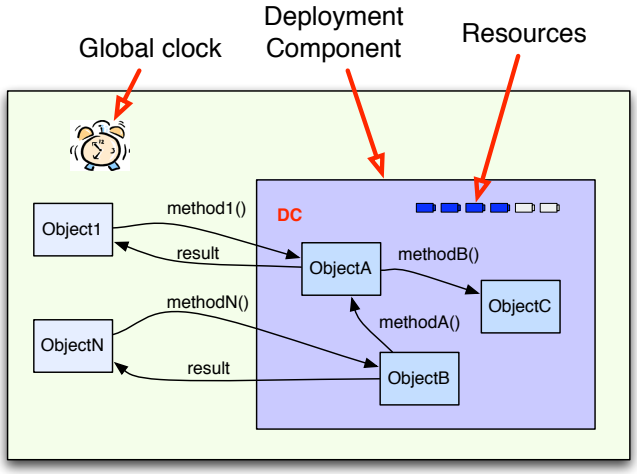

Figure 4.1: Client objects (`Object1` and `ObjectN`) interacting with objects deployed on a deployment component `DC` in ABS.

capacity ("CPU"). We explain below the linguistic primitives implemented in the ABS interpreter reflecting deployment components with abstract processing resources. In the papers attached to the deliverable, we have pushed the investigation a bit further; in particular, by considering dynamic deployment variability "in time" by means of dynamic load balancing based on either the reallocation of (virtual) resources [33] or on object mobility [36], and deployment components which are restricted in memory [4].

4.2.2 Functional Level

A new built-in function, `thisDC()`, returns the deployment component associated with the object that is currently executing. This function always returns a non-`null` deployment component.

4.2.3 Concurrent Object Level

Creating Deployment Components. A new deployment component is created via the `new` statement:

```
1 DeploymentComponent dc = new DeploymentComponent("Server", set[CPUCapacity(50)]);
```

The first argument to the constructor is the name of the new deployment component, which serves as documentation for the modeler and has no semantic meaning. The second argument is a set of resources associated with the new deployment component. At the moment, these can be `CPUCapacity(Int)`, to model its execution capacity, and `URL(String)` to indicate its location.[1]

Associating New Cogs with a Deployment Component. A new cog is associated with a specific deployment component via a `DC:` annotation to the `new cog` statement creating the cog:

```
1 [DC: dc] Server s = new cog Server();
```

The new server `s` will run under the resource constraints of deployment component `dc`. The expression `thisDC()` will return a reference to `dc` in all processes running on object `s` and any object it creates without another specific `DC:` annotation. That is, new objects created without a `DC:` annotation inherit the deployment component of the creating object.

[1] Location does not contribute any semantics currently, but can be used in future distributed implementations of ABS to, e.g., specify the machine where the new object should be created.

```
1  module ResourceExample;
2  import *from ABS.DC;
3  class Server {
4    Time created = now();
5    Time finished = created;
6    Unit process () {
7      [Cost: 5] skip;
8    }
9    Unit run () {
10     Int i = 0;
11     while (i < 10) {
12       this.process();
13       i = i +1;
14     }
15     finished = now();
16   }
17 }
18 {
19   DeploymentComponent dc = new DeploymentComponent("Example", set[CPUCapacity(5)]);
20   [DC: dc] new cog Server();
21 }
```

Figure 4.2: Deployment component example

Expressing Run-time Costs. A statement can be associated with a specific *cost of execution* via the `Cost:` annotation:

```
1  [Cost: 5 *x] listOfData = makeList(x);
```

Executing the above statement will cost a certain number of resources, as described in the next subsection.

Default Run-time Costs. Executing a statement that has no `Cost:` annotation costs a default amount of resources that can be selected at compile-time with the `-defaultcost` parameter to the `generateMaude` script. For example, compiling with the parameter `-defaultcost=1` results in execution costs of 1 for all statements that have no explicit cost annotation.

The default default cost value, when compiling without `-defaultcost`, is 0 (no cost).

Running the model of Figure 4.2 shows the expected result: in the absence of any other object running within `dc`, the `Server` object can consume all five resources that are available per time unit. Consequently, its activity starts at time 0 and ends at time 9, having executed one invocation of its `process` method per time interval. Adding another object to `dc` will cause different timing behavior to emerge.

4.3 Semantics of Resource-Restricted Execution

During time advance, the available CPU resources of all deployment components containing CPU resources are refilled to their original value. This happens atomically at the same time that deadlines and `await duration` conditions get decremented. Unused resources do not "carry over" into new time periods; this models the behavior of real CPUs which can execute a certain number of statements per time unit or stay idle, forfeiting their processing capacity.

Resource-aware semantics is linked to the semantics of Real-Time ABS in order to get abstract time results; i.e., results related to the abstract notions of time and resource used in the model. Subsection 2.2.1 of Chapter 2 summarizes the semantics of clock advance for a basic, time-aware model. The conditions for

$$\frac{cn\ cl(t) \xrightarrow{!} cn'\ cl(t) \quad cn' \xrightarrow{!}_\tau cn''}{\{cn\ cl(t)\} \to_r \{cn''\ cl(t+1)\}}\ (\textsc{RunToCompletion}) \qquad \frac{n < u}{dc(n,u) \to_\tau dc(u,u)}\ (\textsc{Reset})$$

$$\frac{[\![e]\!]^t_{\sigma \circ l} = 0}{o(\sigma, \{l|[\texttt{cost:}e]s\}, q)\ cl(t) \to o(\sigma, \{l|s\}, q)\ cl(t)}\ (\textsc{RestrictedExec1})$$

$$\frac{0 < c \qquad 0 < n \qquad \text{thiscomp}(o) = dc \qquad [\![e]\!]^t_{\sigma \circ l} = c}{o(\sigma, \{l|[\texttt{cost:}e]s\}, q)\ dc(n,u)\ cl(t) \to o(\sigma, \{l|[\texttt{cost:}c-1]s\}, q)\ dc(n-1,u)\ cl(t)\}}\ (\textsc{RestrictedExec2})$$

Figure 4.3: A reduction semantics for timed resource-restricted execution.

advancing the clock (i.e., no process is eligible for execution, no method invocation message is "in flight") are still valid. Deployment components add another constraint for executability:

> An executing process blocks the cog it is executing on if the cost of executing the statement exceeds the available resources in the deployment component.

The reduction rules in Figure 4.3 illustrate the basic idea of how resource management and time advance interact in the ABS interpreter (for simplicity, the presentation only considers a discrete time model). We represent an object by the term $o(\sigma, p, q)$, where o is the identity, p is a process of the form $\{l|s\}$, σ and l are substitutions binding program variables to values (the fields of the object and the local variables of the method activation, resp.), and s is the statement list of the method activation. We represent the system clock by the term $cl(t)$, where t represents the time. Finally, we represent a deployment component by a term $dc(n,u)$, where dc is the identifier, n are the currently available resources of the deployment component, and u are allocated resources on the deployment component. A *configuration* cn consists of a multiset of objects, messages and futures (as for standard ABS), deployment components, and exactly one clock. A system *state* $\{cn\ cl(t)\}$ consists of a configuration and a single clock.

Rule RunToCompletion formalizes the run-to-completion semantics adopted for Real-Time ABS; the reduction relation $\to_r$ captures the reduction of the system states $\{cn\ cl(t)\}$ between observable points in time. Between two observable points in time, the standard ABS reduction relation $\to$ is applied to the configuration $cn\ cl(t)$ until it reaches normal form. (Otherwise, the computation is infinitely fast.) At this point, the system is blocked and time may advance. The effect of time advance is captured by a reduction relation $\to_\tau$. With respect to the resources in the system state, deployment components are replenished; i.e., their available resources are reset to their allocated resources. This way, the resources of a deployment component are allocated for between two observable points in time, which corresponds to processing capacity in our context. Rule Reset captures the replenishment of a deployment component: the consumed resources are added to the available resources.

In order to capture resource consumption, the standard reduction relation $\to$ of ABS is extended with extra rules to cover the case of annotated statements. Thus, rule RestrictedExec1 expresses that a statement with cost 0 reduces to a statement without an annotation. Consequently this removes the resource restriction on evaluating the statement (which at this point happens in zero time). Rule RestrictedExec2 expresses a small-step reduction perspective on resource cost; thus, different objects may partly reduce the cost of their next execution step until the point where the evaluation happens instantaneously. The rule captures that a complex expression can still be evaluated if the cost of a single step surpasses the resources allocated to the associated deployment component, but the execution will then require several time intervals.

Chapter 5

Abstract Failure Models

5.1 Introduction

Abstract Failure Models can be used to represent arbitrary failures in the distributed context of ABS such as network failure or memory exhaustion. In this chapter, we report on a general approach to integrating abstract failure models into concurrent object languages such as the ABS modeling language. This chapter first presents some intuition behind the approach, then we discuss the proposed linguistic extension of ABS, and finally we provide a simple example. This work was done in collaboration between UIO and BOL, and presented in detail in the paper *Fault in the Future* [32].

Concurrent and distributed systems demand flexible communication forms between distributed processes. While object-orientation is a natural paradigm for distributed systems [27], the tight coupling between objects traditionally enforced by method calls may be criticized. Concurrent (or active) objects have been proposed as an approach to concurrency that blends naturally with object-oriented programming. Concurrent objects are reminiscent of Actors [2] and Erlang processes [8]: objects are inherently concurrent, conceptually each object has a dedicated processor, and there is at most one activity in an object at any time. Thus, concurrent objects encapsulate not only their state and methods, but also a single (active) thread of control. In the concurrent object model, *asynchronous method calls* may be used to better combine object-orientation with distributed programming by reducing the temporal coupling between the caller and callee of a method, compared to the tightly synchronized (remote) method invocation model. Intuitively, asynchronous method calls spawn activities in objects without blocking execution in the caller. Return values from asynchronous calls are managed by so-called *futures*. Asynchronous method calls and futures have been integrated with, e.g., Java [37] and Scala [26], and offer a large degree of potential concurrency for deployment on multi-core or distributed architectures.

In ABS, the interaction between objects in different concurrent object groups is modelled using asynchronous method calls and futures [31]. Compared to the approaches discussed above, the polling of futures in ABS is combined with process suspension: a process may be suspended while waiting for the response to a method call, such that other processes in the object may execute in the meantime. This allows a high-level integration of active and reactive behavior in a concurrent object, abstracting from concrete mechanisms for explicit signaling of enabled processes. Thus, concurrent objects in ABS are peers interacting in a distributed setting by means of asynchronous method calls which do not transfer control between the caller and the callee (or the client and the server). However, the standard exception mechanism of object-oriented languages, which propagates errors up the call-chain, is not well-suited to represent failures in the context of asynchronous communicating concurrent objects where the caller activity is not suspended during a remote call.

In the event-driven communication model of Actors and Erlang processes, fault recovery is typically managed by linking processes together [8] or by monitors [1, 49]. These approaches do not address asynchronous method calls and futures. Here, we extend the Java approach [37] with mechanisms for error recovery developed in the context of web services. The approach we take in this work is based on ideas from

s	::=	...	*Standard statements*
	|	`abort` n	(Abort)
	|	`return` e `on compensate` s	(Return)
	|	`on` $x :=$ f`.get do` s $\overline{\texttt{on fail}\ n\ s}$	(Get)
rhs	::=	...	*Standard rhs*
	|	f`.kill`	(Kill)

Figure 5.1: Primitives for error handling

compensation mechanisms in languages for web services, in which failures may occur at either the client side or the server side. A failure which occurs on one side (client or server) is handled on the other side by means of a compensation. This approach allows us to deal with the cancellation of a method call from the client (a so-called `kill`-statement), or with an abort from the server. Futures are used to identify calls, so they provide a natural means to distribute fault notifications and kill requests. In our approach, we use the future associated with the asynchronous method call as a two-way communication channel between the caller and the callee which propagates these failures, so catching a failure on the client side is associated with a `get`-operation on a future. Failure notifications are expressed through `abort`-statements. Aborts have associated user-defined *tags*, such that a statement `abort` n on the server side is compensated by a statement `on fail` n s on the client, where s is the compensation for failure n. This way, we also introduce primitives for defining and invoking *compensations* allowing one to undo already completed method executions. Thus, we obtain a symmetric framework where caller and callee can notify their failure to the partner and manage the incoming notifications. This supports distributed error recovery policies programming.

In our work, we further introduce an extensible type system to ensure that all user-defined failures are captured. The details of this type system will be reported in Deliverable D2.4 [23]. Observe that we have only introduced abstract failures on the server side through the tags associated with `abort`-statements. It is straightforward to extend `kill`-statements in a similar way, to obtain a fully symmetric framework for abstract failure models.

5.2 Abstract Failures in ABS

We propose a symmetric framework for fault handling inside ABS by presenting an extension of the language in which futures are used to return fault notifications and to coordinate error recovery between the caller and callee. The proposed linguistic extension to ABS is given in Figure 5.1 (extending the ABS syntax as given in, e.g., [31]). This section discusses these proposed language primitives.

Futures are used in ABS to return the results of asynchronous method calls, and they uniquely identify those method calls. Thus, they provide a natural means to distribute fault notifications and kill requests. More precisely, the callee of a method can return a fault notification to the caller to signal callee-side failures, while the caller can ask to kill/compensate a previous call. These features support distributed error recovery policies programming.

The caller can ask for termination or compensation of a previous call by performing an operation $x =$ f`.kill` (reminiscent of the `cancel` method of Java futures) on the future f identifying the call, while the callee can signal a failure by executing the `abort` n command (n describes the associated *kind* of failure). If the callee aborts, then it will definitely terminate its activities. Vice versa, if the caller performs $x =$ f`.kill`, it expects the callee to react by executing some compensating activity (in contrast to Java, where the call is just interrupted). The compensating code s is attached to the `return` statement, that we replace with the new command `return` e `on compensate` s. This is the main novelty of our proposal: when a callee successfully terminates, it has not definitely completed its activity, as it will possibly have to perform its compensation activity in the case of failure of the caller. This mechanism is inspired by the compensation mechanisms adopted in service orchestration languages like WS-BPEL [46] or Jolie [25]. A compensation can return a result to the caller: to this aim we use a new future which is freshly created and assigned to x by $x =$ f`.kill`.

It is worth noting that a callee can choose which fault name to return in its corresponding future, identifying the cause of the particular failure through a nominal system of uninterpreted type extensions (i.e., tags which are used to extend the return type of a method). This way, user-defined abstract failures are introduced into the language. In order to allow the caller to properly react to each of these notifications, we had to slightly modify the $x := f.\texttt{get}$ primitive used in ABS to allow the caller to receive the return value. The new construct is $\texttt{on}\ x := f.\texttt{get}\ \texttt{do}\ s\ \overline{\texttt{on fail}\ n_i\ s_i}$, which executes $x := f.\texttt{get}$ as before, but then it executes the statement s if the future f contains a value v, or the statement s_i if f contains a fault name n_i. In the first case the value v is assigned to variable x, otherwise x is unchanged.

Besides presenting the formal definition of the syntax and operational semantics of the new primitives, in [32] we have also discussed how to extend the type system of ABS to ensure that all the faults that may be raised by a method invocation are managed by the caller. The details of the extended type system are given in Deliverable D2.4.

5.3 Example

We illustrate our approach to abstract failure models by modifying the example of Real-Time ABS in Chapter 2 to a client server scenario in which we introduce an explicit failure class for timeout. In this example, the client calls a service `m` on a `Service` object, and provided a deadline to this call. If the call returns normally, the client terminates. If the call returns with a failure with tag `timeout`, the client doubles the deadline and repeats the call. The `Server` class suspends a method activation of method `m` for 5 time intervals, after which it checks whether the deadline has passed. The call is successful if it terminates within the user-provided deadline, otherwise the method activation aborts with a `timeout` failure.

```
1  module FailureExample;
2
3  interface Service { Bool m(); }
4  interface Client { }
5
6  class Server implements Service {
7      Bool m() {
8          await duration(5, 5);
9          if (durationValue(deadline()) ⟩ 0)
10           {return True;}
11           else {abort timeout;}
12     }
13 }
14 class Client implements Client (Service server, Int t) {
15     Void run() {
16       Bool x;
17       [Deadline: Duration(t)] Bool call = server.m(); //normally returns True
18       on x = call.get do skip on fail timeout {t = 2∗t; this!run();}
19     }
20 }
21 {// Main block:
22     Service s = new server();
23     Client c = new client(s,6);
24 }
```

Chapter 6

User-defined Scheduling

6.1 Introduction

This chapter considers variability at the level of scheduling policies. UIO and CWI consider the problem of allowing the application to decide on its own local scheduling policy in order to control the priority of client requests. Scheduling may in general be understood as a two-level problem. At the lowest level, the operating system schedules (virtual) servers for execution. The distributed ABS system exists on a number of servers. Depending on the considered granularity, there may be one server for each deployment component, for each logical component, for each concurrent object group, or for each object. In the presented work, we opt for the most fine-grained approach and consider the scheduling of requests to one object. The main reason for this choice is that scheduling at the level of a single object fits well with the idea of concurrent objects as independent units of concurrency: the scheduler is in charge of selecting which process from the object's process queue to activate when the object is idle. At this level of granularity, it is possible to express highly application-specific scheduling policies in the sense that the scheduler may depend on the state of the object at runtime.

In this chapter, we first explain in Section 6.2 how user-defined scheduling policies may be defined inside the functional level of the ABS modeling language. Next, we propose an integration of scheduling variability into ABS in Section 6.3, in which objects are parametric in their scheduling policy. In practice, we introduce default scheduling policies where the choice is left to the modeling language, and optional annotations to allow the modeller to override these defaults. Different objects of a class may have different scheduling policies. In fact, the choice of scheduling policy may depend on the state of the creator at object creation time. For this work, we use Real-Time ABS (as introduced in Chapter 2) in order to also express timing-dependent scheduling strategies. Further details on user-defined scheduling for Real-Time ABS are given in [11]. Finally, we show in Section 6.4 that the proposed notion of application-level scheduling policies for concurrent objects works well in practice, by means of a proof of concept runtime system for concurrent objects implemented in Java. Here we can measure the performance of concurrent objects with application-level schedulers deployed on a multicore system.

6.2 User-Defined Schedulers in ABS

One of the goals of expressing user-defined schedulers was that scheduling policies be expressible in ABS itself, avoiding the need for a special-purpose language. At the same time, scheduling should be mostly transparent and orthogonal to the functional parts of the model; specifically, a scheduling decision should not have side-effects on the model itself apart from the scheduling decision taken. Consequently, schedulers are expressed using the functional, side-effect-free sublanguage of ABS.

The scheduling policies themselves are expressed as ABS functions. Scheduling decisions are taken using process attributes. We follow the work of the real-time scheduling community (see e.g. [15]) to use process attributes such as deadline, arrival time, priority etc. – the paper [11] contains the details.

```
1 data Pid; //built−in
2 data Process = Process(Pid pid, String method, Time arrival, Duration cost,
3     Duration procdeadline, Time start, Time finish, Bool crit, Int value);
4
5 def Process defaultscheduler(List⟨Process⟩ queue) = head(queue);
6 def Process randomscheduler(List⟨Process⟩ queue) = nth(queue, random(length(queue)));
```

Figure 6.1: ABS process datatype and standard schedulers

During model simulation, whenever a scheduling decision is to be taken, the processes that are ready for execution are *reflected* into ABS datatypes and the scheduler associated with the object is evaluated. Schedulers are simple functions which take a list of processes and return one of them. Figure 6.1 shows the ABS definition of the process datatype and two standard schedulers. Note that the datatype `Pid` does not have a constructor and no ABS program can construct values of this type. This property together with the type-checking of scheduling functions guarantees that all schedulers will return one of the processes that are passed in and cannot construct and try to schedule a "fake" process data structure.

Bjørk et al. [11] contains more complex examples of user-defined schedulers and a discussion of the attributes of the `Process` datatype.

6.3 Scheduling Defaults and Annotations in ABS

There exists a model-wide default scheduling policy for ABS objects, specifically, `defaultscheduler` of Figure 6.1. The modeler can associate schedulers with classes and with single objects. This is done via `[Scheduler: e]` annotations to class definitions and `new` statements, respectively. The following fragment:

```
1 [Scheduler: randomscheduler(queue)]
2 class C implements I {
3   ...
4 }
```

creates a class `C` whose objects are scheduled randomly, for example to run Monte-Carlo-simulations. The first parameter is always named `queue` of type `list<Process>` and represents the object's process queue, which is not otherwise available on the ABS language level.

To create an object of class `C` with a different scheduling policy, the same annotation can be attached to a `new` statement:

```
1   [Scheduler: defaultscheduler(queue)] I o = new cog C();
```

The value for the `Scheduler:` annotation is a complete function call and not only a function name because schedulers can take additional arguments beside the process queue. Values for these additional arguments are taken from the object's state at the time of scheduling. A brief example:

```
1 def Process someScheduler(List⟨Process⟩ queue, Int x) = f(queue, x);
2
3 [Scheduler: someScheduler(queue, attribute)]
4 class C implements I {
5   Int attribute = 42;
6   ...
7 }
```

At each scheduling decision, the value of `attribute` is passed into the scheduler's `x` parameter and can be used to influence the scheduling decision.

6.4 Application-Level Scheduling for Concurrent Objects in Java

CWI and UIO investigate in [45] how concurrent active objects (as in ABS) can be used for high-level scheduling of resources. A concurrent object provides a natural basis for a deployment scheme where each object virtually possesses one processor. We resolve the basic scheduling issue, i.e., which *method* in which *object* to select for execution, by introducing priority-based scheduling of the messages of the individual objects at the application-level itself.

We also propose a tool architecture to deploy applications based on this paradigm. To prototype the tool architecture, we choose Java as it provides low-level concurrency features, i.e., threads, futures, etc., required for multi-core deployment of object-oriented applications. For example, a thread can asynchronously call a method on another thread and use a future variable to get back the result. The tool architecture prototype transforms the constructs for concurrency to their equivalent Java constructs available in the `java.util.concurrent` package. Every active object is transformed to an object in Java that uses a priority manager and scheduler to respond to the incoming messages from other objects. As such, the concurrent object paradigm provides a high-level structured programming discipline based on active objects on top of Java.

Chapter 7

Schedulability Analysis

In general analyzing schedulability of a real time system consists of checking whether all tasks are accomplished within their deadlines. The contributions of this chapter are based on our previous work [29, 30], in which CWI employed automata theory to provide a high-level framework for modular schedulability analysis of concurrent objects. In order to analyze the schedulability of an open system of concurrent objects, we need some assumptions about the real-time arrival patterns of the incoming messages; in our framework, this is contained in the timed automata [6] modeling the *behavioral interface* of the open system. A behavioral interface captures the overall real-time input/output behavior of an object while abstracting from its detailed implementation in its methods; a deadline is assigned to each message specifying the time before which the corresponding method has to be completed. Further, we use timed automata to describe an abstraction of the system of objects itself including its message queues and a given scheduling policy (e.g., earliest deadline first). The analysis of the schedulability of an open system of concurrent objects can then be reduced to model-checking a timed automaton describing the interactions between the behavioral abstraction of the system and its behavioral interface (representing the environment).

When composing a number of individually schedulable open components, the global schedulability of the system can be concluded from the *compatibility* of the components [30], i.e., a message sent by one component should be expected according to the order and timing specified in the behavioral interface of the receiver and furthermore, its deadline may not be smaller than the expected deadline. Formally, we define compatibility in terms of a refinement relation between the composition of the components and the composition of the behavioral interfaces. In this refinement, the messages communicated between components constitute the observable actions. Being subject to state-space explosion, we gave a technique in [30] to test compatibility.

7.1 Conformance Between Real-Time ABS and Timed Automata

Having modeled an open system both in Real-Time ABS and timed automata, conformance between the two models will imply that schedulability results carry over from timed automata models to the ABS models. CWI and UIO introduced in [20] a method to test this conformance with respect to a given behavioral interface. Our method is based on generating a timed trace (i.e., a sequence of time-stamped messages) from the automaton constructed from its behavioral abstraction and interface. Using model-checking techniques we next generate for each time specified in the trace additional real-time information about all possible observable messages. This additional information allows us to find *counter-examples* to the conformance. To do so, we use the Real-Time Maude semantics as a language interpreter to execute the Real-Time ABS model driven by the given trace. Then we look for counter-examples by incrementally searching the execution space for possible timed observations that are not covered in the extended timed trace.

Our overall methodology for the schedulability analysis of an open ABS model consists of the following. To analyze an open system **C**, we need to restrict how its methods are to be called. To this end, we model the real-time pattern of incoming messages in terms of a timed automaton (called the behavioral interface **B** of the ABS model). Sine an ABS model can be quite detailed and complicated, we develop an automata

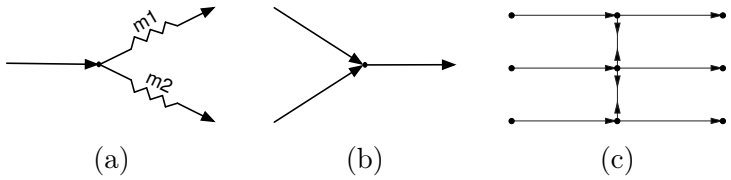

Figure 7.1: More Reo connectors

abstraction of its overall real-time behavior, called **A**. In principle **A** can be automatically generated by translating each method of **C** into a timed automaton [18]. However, to achieve a better abstraction, we develop **A** manually on the basis of sequence diagrams, which describe the observable behavior of the ABS model.

We analyze the schedulability of the product of the abstraction **A** and the given behavioral interface **B** (in, for example, UPPAAL [9]). By establishing conformance between the Real-Time ABS model **C** and its timed automaton abstraction **A**, we show that the schedulability results carry over to **C**. We define this refinement with respect to the given behavioral interface **B** in terms of inclusion of the timed traces of observable actions. we check the conformance by testing in Real-Time Maude as explained below.

We denote by $O(\mathbf{A} \parallel \mathbf{B})$ the set of timed traces of observable actions of the product of the timed automata **A** and **B**. Given any timed trace θ from **B**, *Tester*(θ) denotes an ABS class which implements θ, i.e., it sends to **C** the messages specified in θ at the given times. We define the conformance relation $\mathbf{C} \leq_{\mathbf{B}} \mathbf{A}$ in terms of trace inclusion by

$$O(\mathbf{C}, \mathit{Tester}(\theta)) \subseteq O(\mathbf{A} \parallel \mathbf{B}),$$

for every timed trace of observable actions θ taken from **B**, where $O(\mathbf{C}, \mathit{Tester}(\theta))$ denotes the set of timed traces of observable actions generated by the Real-Time Maude semantics of the ABS model **C** driven by θ.

7.2 Coordinated Concurrent Objects

The general timed automata based framework for schedulability analysis assumes that concurrent objects communicate directly. However, it is beneficial in many cases to separate the computation and coordination concerns by using an exogenous coordination language. In such cases, the schedulability of a system depends also on the network and connections of the objects. In the paper [42], CWI reports on extending the schedulability framework with the coordination language Reo [7] to enable exogenous coordination of active objects. Reo can be used as a "glue code" language for compositionally building connectors that orchestrate the cooperation between components or services in a component-based system or a service-oriented application. An important feature of Reo is that it allows for anonymous communication, i.e., the sender of a message does not need to know the recipient; instead the Reo connector will forward the message to the proper receiver.

With Reo, individually schedulable objects can be used as off-the-shelf modules in a wider variety of network structures. This requires a new compatibility check for our analysis that incorporates the Reo connectors. Our extension preserves the asynchronous nature of the objects, therefore the Reo connectors must have a buffer at every input/output node, which may lead to state-space explosion. To avoid this problem, we provide techniques to optimize the analysis by reusing internal actor buffers in the Reo connectors that are single-input and/or single-output. Although this may seem a strict restriction on the use of Reo, many useful connectors can still be used. Another example of such connectors is shown in Figure 7.1.(a). In this example, the client actor requires two services $m1$ and $m2$ (say 'BookFlight' and 'BookHotel') but there is no server actor that can provide both. The connectors in this figure can be used to connect such a client to two servers each providing one of these services. In this connector filter channels are used which may pass the incoming data only if it matches the pattern provided and thus e.g. distinguishing $m1$ and $m2$. The replies from the two servers can be simply merged using a merger as shown in Figure 7.1.(b).

Although applying a multiple-input, multiple-output connector may in general require an extra buffer at its input, this can be avoided again in several kinds of connectors, which need to be considered individually.

Another example where we can optimize the implementation is a barrier synchronizer, shown in Figure 7.1.(c). A barrier synchronizer delays the messages from the fast client actors until all inputs are ready and only then forwards them to their destinations. In this connector, the destination actor for each input port is statically known; therefore, the buffer of that actor can be used to store messages on the respective input port. In any case, we assume coordination and data flow by Reo happens in zero time.

7.3 Communication Delays

The paper [42] further explains the effect of communication delays on the schedulability of a distributed system. The communication medium between every pair of objects is modeled abstractly by a fixed delay value, called their *distance*. CWI first describes how to implement the effect of delay on messages in an efficient manner with respect to schedulability analysis, and secondly how to extend the compatibility check to take message delays into account. The latter is non-trivial because sending and receiving messages do not happen at the same time any more. Nevertheless, this complication can be hidden from the end user by implementing it in an automatic test-case generation algorithm for compatibility check. Assuming that coordination takes place locally and with no delay, coordination with Reo and communication delays will be orthogonal and can be combined.

We assume a fixed distance for communications between every pair of actors. This is a reasonable assumption if the communication medium between the actors is fixed for all messages. Therefore, the delays in the whole network can be modeled as a matrix; this matrix will be symmetric if we assume the uplink and downlink connections have the same properties. For example, in system composed of a client C and a server S (see Figure 7.2), we assume the distance 1 between the client and the server. The distance of an actor to itself is then zero.

$$\begin{array}{c} \\ C \\ S \end{array} \begin{array}{c} \begin{array}{cc} C & S \end{array} \\ \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \end{array}$$

Figure 7.2: The distance matrix.

Extension of the actor framework with network delays properly addresses the following concerns:

1. The time difference since a message is sent and is executed (at receiver) cannot be smaller than the distance between the sender and the receiver.

2. The deadline associated to each message (specified by the sender) will also include the network delay.

3. The modularity of the analysis techniques will be preserved.

7.4 Decidability for Multiple Processors and Preemptive Scheduling

As we described in this section, we use automata abstractions in order to perform schedulability analysis on RT-ABS models. It is necessary to know in which cases such analysis is decidable. In the paper [28], CWI studies decidable settings of nonpreemptive to preemptive schedulers on single and multiple processor systems while tasks (corresponding to methods in an ABS model) are specified themselves as timed automata. This work is in line with [29,30] that use timed automata for specifying task details in actor-based and concurrent objects settings.

Decidability issues have been studied for different settings of single- and multi-processor systems [24,38]. It is shown that for nonpreemptive schedulers, this problem is decidable; whereas, for preemptive schedulers in general it is undecidable. In certain cases, for example if a task is merely represented by an exact execution time, the problem becomes decidable. Modeling a task simply by an execution time is not enough in many

cases; e.g., if tasks need to generate sub-tasks, if there are dependencies between tasks, or if there are shared resources other than the processor.

In our setting, an object has a processor for executing its tasks. A new task on an object is triggered by receiving a message which requires the execution of the corresponding method. The usage of an object is abstractly defined in its behavioral interface, which is in fact an extension of task automata (because tasks are specified here). Jaghoori et al. [29, 30] consider a nonpreemptive single processor setting. In ABS, there is the possibility of voluntarily releasing the processor during execution of a method, which gives rise to a *cooperative scheduling* paradigm. Cooperative scheduling is in essence nonpreemptive and therefore it is not suitable for an uncontrolled environment, e.g., a general purpose operating system, but it is very powerful if used with care for example in embedded systems design; a task is not preempted in an unstable state whereas all tasks are willing to yield the processor when safe.

Our first contribution is extending cooperative scheduling to preemptive with minimum inter-preemption delays. This assumption is in reality not too restrictive because there is always such a delay as a multiple of the processor clock speed. By encoding this in a decidable class of timed automata, we show that this problem is decidable. Since we allow task behavior to be specified in timed automata, in principle we need the power of stopwatches to be able to model preemption, which is undecidable in general. To be able to model preemption, we restrict task specification automata to use only one clock which in turn cannot be reset by the task itself; nonetheless, tasks can still generate other tasks and subtasks.

We require that there is a minimum delay *delta* between every two preemptions, and that *delta* should be a natural number. The execution time of tasks however need not be an exact integer value, e.g., a task could be modeled as a timed automaton that finishes between 2 and 3 time units. Furthermore, our model of the scheduler can start the next task immediately; i.e., for the above example, it does not have to wait for 3 time units to start the next task.

As our second contribution, we extend the schedulability analysis to a multiprocessor setting and show that schedulability remains decidable. It covers both cooperative and preemptive scheduling. We consider two settings for multiprocessing: shared and separate queues for different processors. The shared queue setting for nonpreemptive scheduling has been studied by CWI and UIO [20]. Further, we study multiple queues for multiple processors with or without task migration. We show a general technique for modeling a load balancing strategy between different processors.

Chapter 8

Concluding Remarks

In this deliverable, we have reported on our work with deployment variability in ABS. Deployment variability is concerned with the lowest levels of a flexible architecture upon which to base ABS models. The HATS Description of Work (DoW) emphasizes linguistic primitives for modeling aspects of the underlying deployment architecture such as concurrency, distribution, scheduling policies, and failure.

We have added these features to ABS in a way which is at the same level of abstraction as the language itself. Further, the extensions are incremental to the behavioral aspect of the models; i.e., deployment decisions can be added at any stage in the development of the model. In particular, these decisions can to a large extent be externalized into separate features, for example in terms of deployment architectures, resource parameters, and optional annotations.

In this deliverable we have focused on "deployment variability in space", which is the theme of the work package for this deliverable in HATS. In many cases, the approaches reported here can be extended quite naturally to "variability in time". For the software components reported in Chapter 3, runtime reconfiguration has in fact been the main motivation for this (still ongoing) work. For the deployment components in Chapter 4, we have included papers researching runtime reconfiguration, although it has not been directly reported in the deliverable. It would be an interesting line of research to pursue temporal variability for other areas of research related to deployment architectures, such as user-defined schedulers and abstract failure models.

We have chosen not to report on the contribution of FRH in this deliverable, although they have participated in the activities of HATS Task 2.1. They are currently exploring the possibilities of deployment variability in the context of the FRH case study. The outcomes of this exploration will be duly reported in WP5.

Bibliography

[1] G.A. Agha and R. Ziaei. Security and fault-tolerance in distributed systems: an actor-based approach. In *Proc. of CSDA'98*, pages 72–88. IEEE Computer Society Press, 1998.

[2] Gul Agha and Carl Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In *Research Directions in Object-Oriented Programming*, pages 49–74. MIT Press, 1987.

[3] Wolfgang Ahrendt and Maximilian Dylla. A system for compositional verification of asynchronous objects. *Science of Computer Programming*, 2010. In press.

[4] 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, *Proc. 17th Intl. Symposium on Formal Methods (FM 2011)*, volume 6664 of *LNCS*, pages 353–368. Springer-Verlag, 2011.

[5] OSGi Alliance. *OSGi Service Platform, Release 3*. IOS Press, Inc., 2003.

[6] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[7] Farhad Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14:329–366, 2004.

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

[9] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. Uppaal 4.0. In *QEST*, pages 125–126. IEEE Computer Society, 2006.

[10] Nina T. Bhatti, Matti A. Hiltunen, Richard D. Schlichting, and Wanda Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Trans. Comput. Syst.*, 16(4), 1998.

[11] Joakim Bjørk, Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. User-defined schedulers for real-time concurrent objects. Submitted for publication.

[12] Joakim Bjørk, Einar Broch Johnsen, Olaf Owe, and Rudolf Schlatte. Lightweight time modeling in Timed Creol. *Electronic Proceedings in Theoretical Computer Science*, 36:67–81, 2010. *Proc. 1st Intl. Workshop on Rewriting Techniques for Real-Time Systems (RTRTS 2010)*.

[13] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quema, and Jean-Bernard Stefani. The Fractal Component Model and its Support in Java. *Software - Practice and Experience*, 36(11-12), 2006.

[14] Lucas Cardelli and Andrew D. Gordon. Mobile Ambients. *Theoretical Computer Science, vol. 240, no 1*, 2000.

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

[16] 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 *LNCS*, pages 417–457. Springer-Verlag, 2011.

[17] Geoff Coulson, Gordon Blair, Paul Grace, Ackbar Joolia, Kevin Lee, and Jo Ueyama. OpenCOM v2: A Component Model for Building Systsms Software. In *Proceedings of IASTED Software Engineering and Applications (SEA '04)*, 2004.

[18] Frank de Boer, Tom Chothia, and Mohammad Mahdi Jaghoori. Modular schedulability analysis of concurrent objects in Creol. In *Proc. Fundamentals of Software Engineering (FSEN'09)*, volume 5961 of *LNCS*, pages 212–227. Springer-Verlag, 2009.

[19] Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In Rocco de Nicola, editor, *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 316–330. Springer-Verlag, March 2007.

[20] Frank S. de Boer, Mohammad Mahdi Jaghoori, and Einar Broch Johnsen. Dating concurrent objects: Real-time modeling and schedulability analysis. In Paul Gastin and François Laroussinie, editors, *Proc. 21st Intl. Conf. on Concurrency Theory (CONCUR)*, volume 6269 of *LNCS*, pages 1–18. Springer-Verlag, September 2010.

[21] Evaluation of core framework, March 2011. Deliverable 5.2 of project FP7-231 620 (HATS), available at `http://www.cse.chalmers.se/research/hats/sites/default/files/Deliverable52.pdf`.

[22] Full ABS Modeling Framework, March 2011. Deliverable 1.2 of project FP7-231 620 (HATS), available at `http://www.hats-project.eu`.

[23] Types for Variability, 2012. Deliverable 2.4 of project FP7-231620 (HATS). To appear.

[24] Elena Fersman, Pavel Krcal, Paul Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149–1172, 2007.

[25] Claudio Guidi, Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. Dynamic error handling in service oriented applications. *Fundamenta Informaticae*, 95(1):73–102, 2009.

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

[27] International Telecommunication Union. Open Distributed Processing — Reference Model parts 1–4. Technical report, ISO/IEC, Geneva, July 1995.

[28] Mohammad Mahdi Jaghoori. From nonpreemptive to preemptive scheduling: from single-processor to multi-processor. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC'11, pages 717–722. ACM, 2011.

[29] Mohammad Mahdi Jaghoori, Frank S. de Boer, Tom Chothia, and Marjan Sirjani. Schedulability of asynchronous real-time concurrent objects. *J. Logic and Alg. Prog.*, 78(5):402 – 416, 2009.

[30] Mohammad Mahdi Jaghoori, Delphine Longuet, Frank S. de Boer, and Tom Chothia. Schedulability and compatibility of real time asynchronous objects. In *Proc. RTSS'08*, pages 70–79. IEEE Computer Society Press, 2008.

[31] 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 *LNCS*, pages 142–164. Springer-Verlag, 2011.

[32] Einar Broch Johnsen, Ivan Lanese, and Gianluigi Zavattaro. Fault in the future. In Wolfgang De Meuter and Gruia-Catalin Roman, editors, *Proc. 13th International Conference on Coordination Models and Languages (COORDINATION 2011)*, volume 6721 of *LNCS*, pages 1–15. Springer-Verlag, 2011.

[33] 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, *Proc. International Conference on Formal Engineering Methods (ICFEM'10)*, volume 6447 of *LNCS*, pages 646–661. Springer-Verlag, November 2010.

[34] 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, *Proc. International Conference on Formal Verification of Object-Oriented Software (FoVeOOS'10)*, volume 6528 of *LNCS*, pages 46–60. Springer-Verlag, 2011.

[35] Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Integrating aspects of software deployment in high-level executable models. In *Norwegian Informatics Conference (NIK'11)*, pages 195–206. Tapir Akademisk Forlag, 2011.

[36] 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, *Proc. 8th International Symposium on Formal Aspects of Component Software (FACS 2011)*, LNCS. Springer-Verlag, 2012. To appear.

[37] *JSR166: Concurrency utilities*. `http://java.sun.com/j2se/1.5.0/docs/guide/concurrency`.

[38] Pavel Krcal, Martin Stigge, and Wang Yi. Multi-processor schedulability analysis of preemptive real-time tasks with variable execution times. In *Proc. Formal Modeling and Analysis of Timed Systems*, volume 4763 of *LNCS*, pages 274–289. Springer-Verlag, 2007.

[39] Michael Lienhardt, Ivan Lanese, Mario Bravetti, Davide Sangiorgi, Gianluigi Zavattaro, Yannick Welsch, Jan Schäfer, and Arnd Poetzsch-Heffter. A component model for the ABS language. 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 *LNCS*, pages 165–183. Springer-Verlag, 2011.

[40] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable. Building Reliable, High-Performance Communication Systems from Components. In *Proceedings of the 1999 ACM Symposium on Operating Systems Principles*, Kiawah Island, SC, December 1999.

[41] Hugo Miranda, Alexandre S. Pinto, and Luis Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *21st International Conference on Distributed Computing Systems (ICDCS 2001)*. IEEE Computer Society, 2001.

[42] Ólafur Hlynsson Mohammad Mahdi Jaghoori and Marjan Sirjani. Networks of real-time actors: Schedulability analysis and coordination. In Farhad Arbab and Peter Ölveczky, editors, *Proc. 8th International Symposium on Formal Aspects of Component Software (FACS 2011)*, LNCS. Springer-Verlag, 2012. To appear.

[43] Fabrizio Montesi and Davide Sangiorgi. A model of evolvable components. In Martin Wirsing, Martin Hofmann, and Axel Rauschmayer, editors, *Trustworthly Global Computing*, volume 6084 of *LNCS*, pages 153–171. Springer-Verlag, 2010. 10.1007/978-3-642-15640-3_11.

[44] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. In *ACM Symposium on Operating Systems Principles*, 1999.

[45] Behrooz Nobakht, Frank S. de Boer, Mohammad Mahdi Jaghoori, and Rudolf Schlatte. Programming and deployment of active objects with application-level scheduling. In *Proceedings of the 2012 ACM Symposium on Applied Computing (SAC)*. ACM Press, 2012. To appear.

[46] Oasis. *Web Services Business Process Execution Language Version 2.0*. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`.

[47] Alan Schmitt and Jean-Bernard Stefani. The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In *Global Computing*, volume 3267 of *LNCS*. Springer-Verlag, 2005.

[48] Sun Microsystems. JSR 220: Enterprise JavaBeans, Version 3.0 – EJB Core Contracts and Requirements, 2006.

[49] Nalini Venkatasubramanian and Carolyn L. Talcott. Reasoning about meta level activities in open distributed systems. In *Proc. PODC'95*, pages 144–152. ACM Press, 1995.

Glossary

Terms and Abbreviations

ABS Abstract Behavioral Specification language. An executable class-based, concurrent, object-oriented modeling language based on Creol, created for the HATS project.

COG Concurrent Object Group, the unit of parallelism in ABS.

Creol A precursor language to ABS, where the concurrency model of ABS with concurrent objects, asynchronous method calls, and cooperative scheduling, was developed.

Core ABS The behavioral functional and object-oriented core of the ABS modeling language.

COSTA A cost analysis tool for ABS, developed by UPM.

Deployment component A modelling abstraction for deployment choices, restricting the execution capacity of different parts of ABS models.

Deployment architecture The architectural description of a model; i.e., how COGs are allocated to deployment components.

Deployment scenario A deployment architecture with given amounts of resources for the different deployment components.

Deployment variability Changes in the deployment scenarios, either in the deployment architecture or in the resources allocated to different deployment components.

Abstract failure A user-introduced failure without a specific interpretation in the model.

Functional level of ABS The part of the ABS language that deals with side-effect-free expressions.

Real-Time ABS An extension to ABS that adds time, durations and deadlines to the model.

Scheduling The act of choosing one of a set of processes for execution.

Software Components A modelling abstraction reflecting the logical units of composition, which provides isolation, mobility, and data-flow reconfiguration capacities.