Project No: 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 D3.5

Autonomous Evolving Systems

Due date of deliverable: (T0+48)

Actual submission date: 1 March 2013

Start date of the project: 1st March 2009

Duration: 48 months

Organisation name of lead contractor for this deliverable: KTH

Integrated Project supported by the 7th Framework Programme of the EC

<table>
<thead>
<tr>
<th>Dissemination level</th>
</tr>
</thead>
<tbody>
<tr>
<td>PU Public</td>
</tr>
<tr>
<td>PP Restricted to other programme participants (including Commission Services)</td>
</tr>
<tr>
<td>RE Restricted to a group specified by the consortium (including Commission Services)</td>
</tr>
<tr>
<td>CO Confidential, only for members of the consortium (including Commission Services)</td>
</tr>
</tbody>
</table>
Executive Summary:
Autonomous Evolving Systems

This document summarizes deliverable D3.5 of project FP7-231620 (HATS), an Integrated Project supported by the 7th Framework Program 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.

This deliverable reports on several lines of work that explore the use of the ABS language and framework to support software evolvability, software adaptation, and auto-configuration at runtime, with the goal of producing software systems that evolve in a goal-driven and, as far as possible, autonomous manner.

Two lines of work are followed, on runtime adaptability in networked models with explicit routing, focusing on performance aspects, and on runtime adaptability based on product line engineering, focusing on software configuration aspects.

The first line of work explores the foundations of efficient runtime adaptability by proposing ABS-NET, a new model for object mobility where programs execute on top of a network model where routing is explicitly integrated into the abstract language runtime. This “network semantics” uses a concept of location independent routing to obtain an execution model where a lot of the overhead due to object and task mobility in existing approaches is eliminated. The deliverable presents a network semantics for several sublanguages of core ABS and proves their correctness by establishing soundness and full abstraction in relation to a standard semantics based on rewrite logic.

A distributed simulation framework for ABS-NET has been developed, allowing to simulate network and program configurations of reasonable size (> 100 objects on < 100 network nodes). The simulation framework is exploited to experimentally study the performance of ABS-NET using different load balancing and performance tuning regimes on a selection of basic network and application models, using different network topologies, and a selection of program examples that perform distributed computation in different types of runtime configuration such as stars, rings, trees, and distributed hash tables (DHTs).

The second line of work concerns dynamic adaptation of ABS-based software product lines (SPL’s). Earlier work in HATS has shown how to statically reconfigure SPL models by taking an ABS core model and a set of delta modules and flattening them to obtain an executable core ABS model of the end product. We add support for runtime product reconfiguration to ABS by adding a dynamic representation of product specifications and delta modules and by deferring the flattening process to the runtime. Product reconfiguration takes the runtime representation of a product and applies a set of dynamic deltas to obtain a different product of the SPL. To support this the ABS language has been extended with constructs for modeling software variability using SPL engineering principles focusing on dynamic aspects. Furthermore an illustrative example is presented.

List of Authors

Mads Dam (KTH)
Andreas Lundblad (KTH)
Radu Muschevici (KUL)
Karl Palmskog (KTH)
Thiago Henrique Burgos de Oliveira (FRG)
Contents

1 Introduction 5
   1.1 Performance Adaptation 5
   1.2 ABS-NET 6
      1.2.1 microABS-NET and milliABS-NET 6
      1.2.2 Experiments 7
   1.3 Dynamic Software Product Lines 7
   1.4 Deviations from the DoW 7
   1.5 List of Papers Comprising Deliverable D3.5 8

2 Location Independent Routing for Process Networks 9
   2.1 Introduction 9
   2.2 microABS 11
   2.3 Network Semantics: Runtime Configurations 15
   2.4 Correctness 20
   2.5 Discussion 23

3 Efficient Routing of Futures in Object Networks 25
   3.1 Introduction 25
   3.2 mABS 26
   3.3 Reduction Semantics 27
   3.4 Network Semantics 28
      3.4.1 milliABS-NET Runtime Syntax 29
      3.4.2 Reduction Semantics 30
   3.5 Normal Forms 31
      3.5.1 Normalization 32
   3.6 Correctness 33
   3.7 Scheduling 36

4 ABS-NET: Fully Decentralized Runtime Adaptation for Distributed Objects 37
   4.1 Introduction 37
   4.2 Core ABS Adaptation 37
   4.3 Network Model and Semantics 38
      4.3.1 Operational Semantics 39
      4.3.2 Assumptions for Adaptability 40
      4.3.3 Node Controller Behaviour 40
      4.3.4 Object Behaviour 40
   4.4 Adaptation 40
      4.4.1 Node Load Balancing 41
      4.4.2 Minimizing Communication and Other Objectives 41
   4.5 Evaluation 42
      4.5.1 Simulator 42
5 Dynamic Software Product Lines

5.1 Introduction .................................................. 50
5.2 Related Work .................................................. 51
  5.2.1 Dynamic Software Update ................................. 51
  5.2.2 Dynamic Software Product Lines ......................... 51
  5.2.3 Related Work within HATS ................................. 52
5.3 Static Variability in ABS ..................................... 53
  5.3.1 Feature model ............................................. 53
  5.3.2 Product selection ......................................... 54
  5.3.3 Core module .............................................. 54
  5.3.4 Delta module ............................................. 54
  5.3.5 Configuration ............................................ 55
5.4 Modeling DSPLs ............................................... 55
  5.4.1 Product selection ......................................... 56
  5.4.2 DSPL Configuration ....................................... 57
  5.4.3 State Transfer ........................................... 59
5.5 Dynamic Reconfiguration ..................................... 59
  5.5.1 Building Self-Adapting Systems ......................... 60
  5.5.2 Self vs. Externally Triggered Adaptation ............... 61
  5.5.3 Adaptive Runtime Environment ......................... 61
5.6 Conclusion ................................................... 61

6 Conclusion ..................................................... 62

Bibliography ..................................................... 63

Glossary .......................................................... 70

A Adaptibility-Simulated Core ABS Programs ................. 72
  A.1 IndependentTasks.abs ..................................... 72
  A.2 Ring.abs .................................................. 72
  A.3 Star.abs .................................................. 74
Chapter 1

Introduction

Software systems are exposed to change for many different reasons:

• Changes in operating conditions. This sort of change can occur, for instance, due to changing performance requirements, changes in application load, or changing operating configurations, for instance due to mobility, changing power requirements, failures, or attacks.

• Changes in execution platform, for instance because hardware or APIs with better performance or new functionality becoming available.

• Changes in the code base, for instance because of patching, code refactoring, or addition or removal of functionality.

To support these different types of changes different approaches are needed. In this deliverable we report on several lines of work that explore the use of the ABS language and the HATS development framework to support software evolvability, software adaptation, and auto-configuration at runtime, with the goal of producing software systems that evolve in a goal-driven and, as far as possible, autonomous manner.

1.1 Performance Adaptation

The first line of work focuses on performance adaptability, specifically the problem of allowing a program to automatically adapt to changes in the configuration of an underlying networked execution platform. This problem has at least the following three dimensions:

• Adaptation mechanisms: First, a set of mechanisms is needed that allows the system to reconfigure in a safe manner. This includes mechanisms such as reconfiguring local execution settings such as priorities and scheduling options, migrating objects and tasks between cores/execution units, and changing memory allocation or channel bandwidth.

• Monitoring, measurement, and instrumentation: Second, a set of mechanisms is needed for controllers and schedulers to determine the system state. This includes routing information, processor and link loads, and other, more global, information such as end to end latencies.

• Adaptation algorithms: Finally, control algorithms are needed that use the adaptation and monitoring artefacts to control execution to meet given systems objectives, here typically performance and safety requirements. Generally the goal is that the system state stays within a desired safe operation perimeter, and such that operation is efficient, i.e., it does not consume more resources than necessary while meeting required performance properties, for instance related to resource utilization or application performance.
Figure 1.1: Control systems view of adaptability

Figure 1.1 fits these components into a familiar control systems setting with the important proviso that “objectives” are (derived from) global systems objectives whereas control decisions are made locally at each processing node, based on local monitored behaviour. This control systems view of adaptability is quite well accepted. Much work in the cloud computing domain, for instance, aims at achieving datacenter scale predictability of latency and bandwidth consumption using dynamic, but centralized control schemes, for problem domains such as VM scheduling, network access, and live migration cf. [45, 44, 55, 2]. The main differentiator with respect to the work reported here is that we take an abstract, semantics-based approach aimed at producing solutions that are fully decentralized, dynamic, and scalable (though we do not claim to have achieved these goals yet).

1.2 ABS-NET

As our main contribution in this part of the deliverable we propose ABS-NET: a new model for object mobility where programs execute on top of a network model where routing is explicitly integrated into the abstract language runtime. This “network semantics” uses a concept of location independent routing to obtain an execution model where a lot of the overhead due to object and task mobility in existing approaches is eliminated. This is important, since existing approaches to object mobility suffer from inefficiencies due to rerouting and message forwarding that typically make them unusable in application scenarios requiring high performance. Our approach could be used to optimize live migration overhead in practical cloud computing settings as well: To the best of our knowledge this type of approach has not yet been attempted.

1.2.1 microABS-NET and milliABS-NET

The network semantics is developed in two steps. In Chapter 2 of the deliverable, we first introduce a small fragment of the core ABS language, microABS-NET, essentially corresponding to a language of asynchronous message passing processes. The microABS-NET language corresponds roughly to the nomadic PICT language by Sewell and his colleagues [78]. The fragment is then enriched in Chapter 3 to capture what is essentially the asynchronous fragment of core ABS, called milliABS-NET. For both these languages, operational semantics are given at an abstract level using a standard rewriting logic approach, in order to determine the reference run-time behaviour of programs when most aspects of physical distribution, including naming, routing, and locality are ignored, similar to the style of semantics considered in other of the HATS project deliverables.

For each of the languages, a network semantics is then proposed that localizes program execution to explicit network nodes, connected pairwise by asynchronous, buffered, point to point links. We then analyze the correctness of the network semantics by proving the equivalence of the two semantics in the absence of a scheduler, i.e., where all runtime choices concerning routing updates, object migration, message delivery, and task scheduling are resolved in a fully nondeterministic fashion. Part of this analysis involves proving convergence and self-stabilization type properties. The correctness results are very strong, in one go establishing
 Deliverable D3.5  Autonomous Evolving Systems

a lot of desirable properties such as absence of deadlocks, eventual delivery of messages, as well as critical consistency properties. Moreover, as we show, the results can be used to easily derive simulation results in the presence of scheduling as well.

1.2.2 Experiments
In Chapter 4 we report on an extension of ABS-NET to the full core ABS language, as described in [56]. We present a network semantics generalizing the microABS-NET and milliABS-NET semantics, but organized a little differently by splitting the runtime state into two components as follows:

1. A language-independent node controller, responsible for monitoring, exchanging state information with neighbours, and adaptation, making scheduling decisions chiefly by determining when and where to migrate objects.

2. A language dependent interpreter layer, responsible for program execution, delegating all functionality concerning control, communication and synchronization, object migration, and scheduling to node controllers.

A distributed simulation framework for ABS-NET has been developed, allowing to simulate network and program configurations of reasonable size (>100 objects on <100 network nodes). The simulation framework is exploited to experimentally study the performance of ABS-NET using different load balancing and performance tuning regimes on a selection of basic network and application models, using different network topologies, and a selection of program examples that perform distributed computation in different types of runtime configuration such as stars, rings, trees, and distributed hash tables (DHTs).

1.3 Dynamic Software Product Lines
In Chapter 5 we present our approach to handle dynamic reconfiguration (runtime adaptation) of ABS code. Dynamic software product reconfiguration is understood as the ability to reconfigure products at runtime, that is, the transformation of a product into another valid product defined by a given software product line (SPL), all without the need to re-compile and redeploy the system. To support this kind of dynamic adaptation, ABS models need to accommodate dynamic changes in their structure and behaviour. Adding this facility complements the static SPL modeling capability of ABS. Static product generation introduced support for configuring a particular SPL product at compile time by taking an ABS core model and a set of delta modules and flattening them to obtain an executable core ABS model of the end product. We add support for runtime product reconfiguration to ABS by adding a dynamic representation of product specifications and delta modules and by deferring the flattening process to the runtime. Product reconfiguration takes the runtime representation of a product and applies a set of dynamic deltas to obtain a different product of the SPL.

The ABS language has been extended with constructs for modeling software variability using SPL engineering principles focusing on dynamic aspects. Furthermore an illustrative example is presented.

If from a technical perspective everything can be changed/adapted at runtime, from the product line perspective only a limited range of runtime changes may make sense and maintain the consistency of a product. In order to avoid breaking product consistency when adapted at runtime, there must be a way to constrain the adaptations that could be performed at runtime. Thus, the possible adaptations performed at runtime must be planned such that the resulting product (after the adaptations are performed) is a consistent product adhering to all feature model constraints and definitions.

1.4 Deviations from the DoW
Task 3.5 as formulated in the DoW focuses on three problem domains, namely functionality, performance, and security. To keep the scope of Task 3.5 manageable it was decided early on to delegate security concerns
regarding adaptability and self-monitoring to the relevant Tasks 4.1 and 3.4. The task has also had a more explorative character than envisaged in the DoW, since the development and analysis of the ABS-NET model turned out to be a larger enterprise than foreseen. For this reason, the ABS-NET simulator got operational too late in the task for larger scale case studies to be possible. Stability and robustness concerns played a smaller role than foreseen in the DoW: Self-stabilization properties implicitly play an important role in the analysis of ABS-NET. However, stability and robustness becomes more central once attention moves beyond performance to security and fault tolerance.

1.5 List of Papers Comprising Deliverable D3.5

This section lists all the papers that comprise this deliverable, indicating how each paper is related to the main text of this deliverable. The papers have not yet been submitted for publication. As requested by the reviewers, the papers are not directly attached to Deliverable 3.5. A version of this deliverable with the papers attached is available on the HATS web site at the following url:


**Paper 1: Location Independent Routing in Process Network Overlays**

This paper [24] introduces the microABS-NET fragment of core ABS and its network semantics using location independent routing, and proves correctness of the network semantics in the sense of barbed equivalence with respect to a semantics given in standard rewriting logic style.

The paper is written by Mads Dam and a conference version is to be submitted during the late winter 2013.

(Download [Paper 1](#))

**Paper 2: Efficient and Fully Abstract Routing of Futures in Object Network Overlays**

This paper [26] extends the results of the previous paper by considering a richer fragment, milliABS, of core ABS which includes futures, and extends the correctness results of [24] accordingly.

The paper is written by Mads Dam and Karl Palmskog and a conference version is to be submitted during the late winter 2013

(Download [Paper 2](#))

**Paper 3: ABS-NET: Fully Decentralized Runtime Adaptation for Distributed Objects**

The paper [25] extends the ABS-NET model to allow interesting test programs in ABS to be implemented, and examines runtime performance adaptation on top of a distributed simulator developed for the ABS-NET platform

The paper is written by Mads Dam, Ali Jafari, Andreas Lundblad, and Karl Palmskog and a conference version is to be submitted during the late winter/early spring 2013

(Download [Paper 3](#))
Chapter 2

Location Independent Routing for Process Networks

2.1 Introduction

The decoupling of applications from their physical realization, intimately tied to the concept of virtualization, is a recurrent theme in the history of computing. Running applications on virtual machines allows many tasks to be performed independently of the physical computing infrastructure. By migrating virtual machine images between physical processors it is possible for a cloud provider to adapt processing and communication load to changing application demands and to changes in the physical infrastructure. In this way applications can, at least in principle, be freed of the burden of resource allocation. That is, it is left to an underlying processing network to determine on which nodes to place which tasks in order to make efficient use of processing resources, while at the same time meeting requirements on response times and processing capacity. If realized, the result is simpler application logic, better service quality, and, ultimately, lower costs for development, operations and management.

The question is how to realize this potential with a minimum of overhead, and in such a way that applications behave in a predictable manner. We examine this question in the context of a rudimentary distributed object language, and propose a formal, networked, runtime semantics of this language with some quite novel features. The goal is a “bare-metal” style of semantics where all aspects of computing and communication are accounted for in terms of local operations that could be directly implemented on top of silicon or, say, a hypervisor such as Xen [9].

A key problem is how to handle object and task mobility in an efficient manner. Since the allocation of objects to nodes is dynamic, some form of application level routing is needed to ensure that messages reach their destinations quickly, and with minimal overhead. Various approaches have been considered in the literature (cf. [78] for a survey):

- One option is to maintain a centralized or distributed database of object locations. Such a database can be used for both forwarding, by routing messages through the forwarding server, and for location querying, by using the database to look up destination object locations. In either case, object location and the location database must be kept consistent, which requires synchronization. Many experimental object mobility systems in the literature use some form of replicated or distributed location databases, cf. [35, 78, 12, 46].

- Another option is for nodes to maintain forwarding pointers, as in the Emerald system [60]. Migration then causes forwarding pointer chains to be extended by one further hop, and some mechanism is typically used to piggyback location update information onto messages, to ameliorate forwarding chain growth. This mechanism is used, for instance, in JoCaml [22].

- Many solutions involve some form of broadcast or multicast search. For instance, an object may use multicasting to find an object if a pointer for some reason has become stale, as in Emerald, or for
service discovery, as in Jini [4].

- Other solutions have been explored too, such as tree-structured DNS-like location directories [84], Awerbuch and Peleg’s distributed directories [7], and Demmer and Herlihy’s arrow protocol [34].

The main source of the difficulties these approaches are designed to solve is the distinction between destination host identifiers (location) and search identifiers (object identity). But, in a fully mobile setting the location at which an object resides has no intrinsic interest. What is of interest is message destination, that an RPC destined for the object with identity $o$ is routed to the location where $o$ resides, and not somewhere else. The address of the destination is not relevant. In other words, rather than routing messages according to IP address, inter object message routing should really be done according to the identity of the destination object rather than an assumed host identity, which might, for all the sender knows, be completely out of date.

This problem is in fact well known in the networking community, and has been the subject of significant attention over the last decade. The idea is to replace the location-based routing of traditional IP networks with location independent schemes that route messages according to names, or content. Names can be flat, unstructured identifiers, as in [14], or they can encode some form of signed content identity, as in content centric networking [53]. The general goal is to devise routing schemes for flat name spaces that are compact, such that routing tables can be represented at each node using space sublinear in the size of the network, and such that path lengths, and hence message latencies, do not grow too far from the optimal. The latter requirement of low stretch, defined as the ratio of route length to shortest path length, precludes the use of both location registers and hierarchical IP-like naming schemes.

The main purpose of this chapter is to show that name-based routing offers a new space for solutions to the object mobility problem with some attractive properties:

- No centralized or decentralized object location database is needed, since the network routing mechanism itself ensures that messages are routed to their proper hosts.

- A whole swathe of software becomes superfluous, which manages address lookups, message forwarding, rerouting, address bookkeeping, and the synchronization overhead between location registers and the migrating objects is eliminated. As a result, the “trusted computing base” of the networked execution platform is significantly reduced, in terms of size, and complexity.

- Traffic overhead is decreased. First, mobility support on top of IP needs to perform routing both at IP and at application level. Name-based routing in effect eliminates the need for IP level routing. Moreover, in steady state the simple distance vector routing scheme used here has stretch 1, so message delivery overhead is minimal (however, distance vector routing is not compact so our scheme does not scale well. We leave this issue for future work).

- Improved robustness: In faulty situations, if connection to the location register is lost, message delivery is impossible (or needs to resort to more costly mechanisms such as broadcasting, as in Emerald or Jini). Routing can be made self-stabilizing and thus able to adapt to any type of disturbance, as long as connectivity is maintained. This allows computation to progress (including delivery of messages and migration of objects) even when the network is under considerable churn.

On the other hand, throwing away network layers 3 and above may seem an excessive price to pay, and the argument above that the cost of IP routing should be taken into account is evidently invalid if IP routing needs to be performed anyway in any realistic implementation. We argue, however, that this does not need to be the case. First, as we have noted, the general architecture of the future internet is currently very much in flux. Second, although we have so far only explored an early prototype simulator [25] built on top of an IP-based overlay, it is perfectly possible today to build large scale non-IP networks with only layer 2

\[^1\]Location has interest as a source of latency, for instance, but that is another matter.

\[^2\]Routing tables in distance vector routing have size $\Omega(n \log n)$
connectivity, sufficient to bootstrap our approach. Third, the simplicity of our approach in comparison to the task of formally verifying, e.g. IP and TCP [13], opens up for the possibility of extending currently ongoing work on formally verified low level software along the lines of seL4 [61] to fully networked operating systems and hypervisors at the device and instruction level. Finally, even if amending the current IP naming schemes turns out to be infeasible, it is still of interest to evaluate the consequences of a much tighter integration between network infrastructure and application level runtimes than what is currently done.

Our study is set in the context of a distributed object language microABS, a small fragment of core ABS [56]. The microABS language includes only a rudimentary set of features, sufficient, however, to allow simple networked programs to be programmed in a natural way, as we illustrate by a couple of small examples. The microABS language is class based, and has features for remote method invocation, object creation, and standard sequential control structures, similar to Sewell et al.'s Nomadic Pict language [78]. Two semantics of microABS are given, one a standard semantics in rewriting logic style which does not take into account aspects related to location, naming, routing, or communication. The second semantics takes these aspects into account by executing objects on top of an arbitrary, but concrete, processor network. The main result of the chapter is to show that the network semantics is sound and fully abstract with respect to the reference semantics. We base the analysis on barbed equivalence [73]. Barbed equivalence requires a witness relation that preserves some set of primitive observations, here remote calls to external objects, in both directions, and is preserved under weak reductions, also in both directions. Barbed equivalence is convenient since the required (unlabelled) reduction relations are easy and fairly uncontroversial to define, both for the reference semantics and for the network semantics. Using a labelled transition semantics particularly at the network level is much more complex, and needs to be subject to a separate justification which is out of scope for the present work.

We structure the presentation as follows: In Section 2.2 we introduce the microABS language, briefly discuss the reference semantics, and introduce the notion of barbed equivalence. The detailed semantics follows the rewriting style of semantics used in many ABS-related works and can be found in [24]. We then turn to the network semantics. In Section 2.3 we first introduce runtime states, or configurations, including routing and network graphs, and the reduction relation. Details concerning well-formedness conditions that need to apply at both the reference (so-called type 1) level and at the network (type 2) level are deferred to the full version. In Section 2.4 we then summarize the soundness and full abstraction proof showing that initial configurations at type 1 and type 2 levels are barbed equivalent. Finally, in Section 2.5 we conclude.

2.2 microABS

The syntax of the object language microABS is presented in Figure 2.1. We use a standard boldface notation for vectors. Thus, e.g. \( \mathbf{x} \) is a vector of variables. Programs are sequences of class definitions, along with global variables \( \mathbf{x} \), and a "main" statement \( s \). The class hierarchy is flat and fixed. Objects have parameters \( \mathbf{x} \), local variable declarations \( \mathbf{y} \), and methods \( \mathbf{M} \). Methods have parameters \( \mathbf{x} \), local variable declarations \( \mathbf{y} \) and a statement body. For simplicity we assume that variables have unique declarations. The definition of expressions \( \epsilon \) is left open, but we require that expressions are side-effect free. The language is untyped. It is

| \( x, y \in \text{Var} \) | Variables | \( e \in \text{Exp} \) | Expression |
| \( P \) | \( \text{CL}(\mathbf{x}, s) \) | Program |
| \( \text{CL} \) | \( \text{class } \mathbf{C}(\mathbf{x})\{\mathbf{y}, \mathbf{M}\} \) | Class definition |
| \( M \) | \( m(\mathbf{x})\{\mathbf{y}, s\} \) | Method definition |
| \( s \) | \( s_1; s_2 \mid x = \epsilon \mid \textbf{skip} \) | Statement |
|  | \( \mid \text{if } e\{s_1\} \text{ else } \{s_2\} \) |
|  | \( \mid \text{while } e\{s\} \mid e!m(\epsilon) \) |
| \( \epsilon \) | \( \epsilon \mid \text{new } \mathbf{C}(\epsilon) \) | Right hand sides |

Figure 2.1: microABS abstract syntax
possible to add types and a notion of well-typedness. However, this does not affect the presentation in any significant way.

Besides standard sequential control structures (the choice of which is largely irrelevant), statements involve a minimal set of constructs for asynchronous method invocation and object creation. Method bodies lack a return statement. All interobject communication is therefore by RPC. Return statements with futures are considered in Chapter 3 of this deliverable. For now, method bodies are simply evaluated to the end at which point the evaluating task is terminated. In the absence of return statements, objects communicate using callbacks in a manner which is not dissimilar to communication in Erlang, as illustrated in the following examples.

Example 2.2.1. A very simple server applying \(\text{foo}\) to its argument is shown in Figure 2.2. We assume a reserved OID \(\text{env}\) with reserved method \(\text{output}\) to be used as a standard output channel.

Example 2.2.2. Just to show that the language is not trivial, the program in Figure 2.3 constructs an object ring with (here) 42 elements. A value is circulated along the ring, computing \(\text{bar}(...)\). Each ring element decrements an iterator \(\text{iter}\) initialized to the original value 42 first received. The final ring element returns the final value to the server, which then finally returns it to the client through the \(\text{output}\) method of object \(\text{env}\).

We give a reduction semantics, the reference, type 1 semantics, for \textit{microABS} in in terms of a transition relation \(cn \rightarrow cn'\) where \(cn, cn'\) are \textit{configurations}, as determined by the runtime syntax in Figure 2.4. For a detailed explanation of the runtime syntax we refer to [24] and highlight only the main points here. Object identifiers, in particular, are \textit{names}, and the \(\pi\)-like binder \textit{bind} is used to bind OIDs. The binder acts as name binding in \(\pi\)-calculus, with similar scope intrusion and extrusion properties. For computations to have observable effects we assume a fixed set Ext of external OIDs with dedicated methods, such as the OID env and the method \textit{output} in Examples 2.2.1 and 2.2.2. Call containers play a special, somewhat subtle role in defining the external observations of a configuration \(cn\). An observation, or \textit{barb}, is a call expression of the form \(o!m(v)\), ranged over by obs. In order to define the observations of a given configuration, we assume a fixed set Ext of external OIDs to which outgoing method calls can be directed. Names in Ext are not allowed to be bound. A barb, then, is an external method call, i.e. a method call to an OID in Ext. Calls that are not external are meant to be completed in usual reduction semantics style, by internal reaction with the called object, to spawn a new task. External calls could be represented directly, without introducing a special container type (which is not present in the core ABS semantics of [50]), by saying that a configuration \(cn\) has barb \(obs = o!m(v)\) if and only if \(cn\) has the shape

\[
\text{bind } o_1.(cn' \ o(o_2, a) \ t(o_2, l, e_1!m(e_2); s)),
\]

(2.1)

where \(e_1(a, l) = o \in \text{Ext}\) (the value of expression \(e\) in object environment \(a\) and method environment \(l\)) and \(e_2(a, l) = v\). However, in a semantics with unordered communication, which is what is assumed of
```java
class Server(){
    serve(from, x){
        c = new Cell();
        c!process(x, self, x)
    }
    return(result){
        from!response(result)
    }
},

class Cell(){
    process(x, root, iter){
        if iter = 0 {
            root!return(y)
        } else {
            c = new Cell();
            c!process(bar(x, iter), root, iter-1)
        }
    }
},
class Client(arg){
    use(server){
        server!serve(self, arg)
    }
    response(y){
        env!output(y)
    }
}[

server = new Server();
client = new Client(42); clien
```
new real-world—or at least not overly artificial—networked software systems.

wlocal: If $x \in \text{dom}(l)$ then $t(o, l, x = e; s) \rightarrow t(o, l[e(a,l)/x], s)$

wfield: If $x \in \text{dom}(a)$ then $o(a, a) t(o, l, x = e; s) \rightarrow o(a, a[e(a,l)/x]) t(o, l, s)$

skip: $t(o, l, \text{skip}; s) \rightarrow t(o, l, s)$

if-true: If $e(a, l) \neq 0$ then $t(o, l, \text{if } e(s_1) \text{ else } s_2; s) \rightarrow t(o, l, s_1; s)$

if-false: If $e(a, l) = 0$ then $t(o, l, \text{if } e(s_1) \text{ else } s_2; s) \rightarrow t(o, l, s_2; s)$

while-true: If $e(a, l) \neq 0$ then $t(o, l, \text{while } e(s_1); s) \rightarrow t(o, l, s_1; \text{while } e(s_1); s)$

while-false: If $e(a, l) = 0$ then $t(o, l, \text{while } e(s_1); s) \rightarrow t(o, l, s)$

Figure 2.5: milliABS reduction semantics part 1

Figure 2.6: Sample microABS reduction rules

draw strong conclusions also in the case a scheduler is added, as we discuss later. Barbed equivalence requires of a pair of equivalent configurations that the internal transition relation $\rightarrow$ is preserved in both directions, while preserving also a set of external observations. Although weaker than corresponding equivalences such as bisimulation equivalence on labelled transition systems, barbed equivalence is nonetheless of interest for the following two reasons:

1. Barbed equivalence offers a reasonable account of observationally identical behaviour on closed systems, i.e., when composition of (in our case) subconfigurations to build larger configurations is not considered because it a) is for some reason not important or relevant, or b) does not offer new observational capabilities.

2. Barbed equivalence can be strengthened in a natural way to contextual equivalence [71] by adding a natural requirement of closure under context composition. Furthermore, a number of works [54] [74] have established very strong relations between contextual equivalence for reduction oriented semantics and bisimulation/logical relation based equivalences for sequential and higher-order computational models.

It is, however, far from trivial to devise a natural notion of context that works at the level of the network semantics introduced later, and such that the notions of context correspond at both the abstract, reference semantics level we consider at present, and at the network level. For this reason the present account based on barbed equivalence is also a natural stepping stone towards a deeper study of the notion of context in real-world—or at least not overly artificial—networked software systems.

Let $\text{obs} = \alpha^!m(\nu)$. The observation predicate $cn \downarrow obs$ is defined to hold just in case $cn$ can be written in the form

$$\text{bind } o.(cn' c(\alpha', m, \nu))$$

The derived predicate $cn \downarrow obs$ holds just in case $cn \rightarrow^* cn' \downarrow obs$ for some $cn'$.  

14
Let now $R$ be a binary relation on type 1 well-formed configurations. We are interested in relations with the following properties:

- **Symmetry**: If $cn_1 R cn_2$ then $cn_2 R cn_1$
- **Reduction-closure**: If $cn_1 R cn_2$ and $cn_1 \rightarrow cn'_1$ then there exists some $cn'_2$ such that $cn_2 \rightarrow^* cn'_2$ and $cn'_1 R cn'_2$
- **Barb preservation**: If $cn_1 R cn_2$ and $cn_1 \downarrow obs$ then $cn_2 \downarrow^* obs$

We call a relation with these three properties a **type 1 witness relation**.

**Definition 2.2.3** (Type 1 Barbed Equivalence). Let $cn_1 \simeq_1 cn_2$ if, and only if, $cn_1 R cn_2$ for some type 1 witness relation $R$.

### 2.3 Network Semantics: Runtime Configurations

The “standard” (type 1) semantics for microABS is quite abstract and does not account for many issues which must be faced by an actual implementation, in particular if the goal is high performance and scalability. For instance:

- The microABS semantics implements a rendez-vous oriented communication model. We want to account for this using a standard buffered asynchronous model.
- Accordingly, calls should be replaced by message passing.
- The microABS semantics has no concept of proximity or name space. Any two objects, regardless of their “location” can without any overhead or search choose to synchronize at any point. Instead, we want a semantics that is **network aware** in the sense that it brings out proximity and location without unduly constraining the model, for instance to a particular naming discipline, or to a centralized name or location lookup service.

Our proposal is to execute microABS objects on a network graph in a fully decentralized and lock free manner where the only means of communication or synchronization is by asynchronous message passing along edges connecting neighbouring nodes, each edge having an associated directional, buffered communication channel. In this section we accordingly introduce a refinement of the standard semantics, a “network semantics”, or type 2 semantics, which adds an explicit network components to the type 1 semantics. The key idea is to use name-based routing, as explained in the introduction. That is, nodes are equipped with explicit routing information allowing messages to be addressed to specific receiving objects, rather than their hosts, which may change. This allows a very simple, fully decentralized, and lock free integration of routing and object migration, as we now begin to demonstrate.

**Nodes and Routing**  The network semantics is presented in rewriting logic style, similar to the type 1 semantics above. We still have configurations $cn$, but these now have a richer structure. We first introduce two new types of container to reflect the underlying network graph, namely **nodes** and **links**. Node containers have the form $n(u, t)$ where $u \in NID$ is a primitive **node identifier**, and where $t$ is an associated routing table. Node identifiers (NIDs) take the place of IP addresses in the usual IP infrastructure. For routing we assume a rudimentary Bellman-Ford distance vector routing discipline [82]. More elaborate and practical routing schemes exist that are better equipped for e.g. disconnected operation, and with better combinations of scalability and stretch. However, for the present purpose, the distance vector scheme is quite adequate. Consequently, a **routing table** $t$ is a partial function associating to the OIDs $o$ “known” to $t$ a pair $t(o) = (u, n)$ where $n$ is the minimum number of hops believed by $t$ to be needed to reach the node hosting $o$ from the current node, and where $u$ is the next hop destination.
Routing Tables  Routing tables support the following operations:

- Next hop lookup, \( \text{nxt}(o,t) = \pi_1(t(o)) \), the first projection of \( t(o) \): In the context of a node \( n(u,t) \), \( \text{nxt}(o,t) \) returns a neighbour \( u' \) of \( u \) to which, according to the current state of \( u \), a message should be sent in order to eventually reach the destination \( o \).

- Update, \( \text{upd}(t,u,t') \): Updates \( t \) by incorporating the routing table \( t' \) belonging to a (neighbouring) node \( u \). The update function is defined thus:

\[
\text{upd}(t,u,t')(o) = \begin{cases} 
\bot & \text{if } o \not\in \text{dom}(t) \cup \text{dom}(t') \\
 t(o) & \text{else, if } o \not\in \text{dom}(t') \\
 (u,\pi_2(t'(o)) + 1) & \text{else, if either } o \not\in \text{dom}(t), \text{ or } \pi_1(t'(o)) = u, \text{ or } t'(o) < \pi_2(t(o)) - 1 \\
 t(o) & \text{otherwise}
\end{cases}
\]

If \( o \) is known to neither the current node nor to \( u \), the distance estimate to \( o \) from the current node is undefined. If it is known to the current node, but not to \( u \), \( u \)'s information is unchanged. If it is known to \( u \), but not to the current node, the estimate from the current node becomes \( 1 \) plus \( u \)'s estimate. Otherwise we may assume that \( u \) is known to both the current node and to \( u \). If the minimal route follows the edge between the current node and \( u \), \( u \)'s distance estimate plus one is the new distance estimate at the current node, regardless of whether the estimate improves on the current estimate or not. Otherwise, if \( u \)'s estimate improves sufficiently on the estimate at the current node, \( u \)'s estimate is incremented and used at the current node. In other circumstances, the distance estimate at the current node is left unchanged.

- Registration, \( \text{reg}(o,u,t) \): Returns the routing table \( t' \) obtained by registering \( o \) at \( u \) in \( t \), i.e.

\[
\text{reg}(o,u,t)(o') = \begin{cases} 
(u,0) & \text{if } o = o' \\
t(o') & \text{otherwise}
\end{cases}
\]

The function \( \text{reg} \) is invoked only when \( u \) is the “current” node.

Links, Queues, and Messages  Nodes are connected by directed edges, or links, of the form

\[ l(u,q,u') \]

where \( u \in \text{NID} \) is the source NID, \( u' \in \text{NID} \) is the sink NID, and where \( q \in Q \) is the associated fifo message queue. Queue operations are standard: \( \text{enq}(msg,q) \) enqueues the message \( msg \) onto the tail of \( q \), \( \text{hd}(q) \) returns the head of \( q \), and \( \text{deq}(q) \) returns the tail of the \( q \), i.e., \( q \) with \( \text{hd}(q) \) removed. If \( q \) is empty \( (q = \varepsilon) \) then \( \text{hd}(q) \) and \( \text{deq}(q) \) are both undefined.

Messages have one of the following three forms:

- \( \text{call}(o,o',m,v) \): A remote call message originating from object \( o \) and addressed to object \( o' \), of method \( m \), and with arguments \( v \).

- \( \text{table}(t) \): A routing table update message. The origin NID is implicit, as the message is dequenced from a link queue with explicit source NID.

- \( \text{object}(cn) \): An object migration message, where \( cn \) is an object closure, as explained below.

Call messages are said to be object bound, and table and object messages are said to be node bound. We define \( \text{dst}(msg) \), the destination of \( msg \) to be \( o' \) for call messages, and \( \text{dst}(msg) = \bot \) in the remaining two cases.
The Network Graph  Nodes and links induce a directed graph structure \( \text{graph}(cn) \) in the obvious way, by taking as vertices the NIDs \( u \) and as edges pairs \((u, u')\) for each link \( l(u, q, u')\). For this to make sense we impose some constraints that apply, from now on, to all “global” configurations \( cn \) in the type 2 semantics.

1. Unique vertices: There is at most one container \( n(u', t) \in cn \) with \( u' = u \).

2. Unique edges: For each source-sink pair \( u, u' \) there is at most one link \( l(u, q, u') \), for some \( q \).

3. Edges connect vertices: If \( l(u, q, u') \in cn \) then \( n(u, t), n(u', t') \in cn \) for some \( t, t' \).

4. Reflexivity: \( \text{graph}(cn) \) is reflexive, i.e., if \( n(u, t) \in cn \) for some \( t \) then \( l(u, q, u) \in cn \) for some \( q \).

5. Symmetry: \( \text{graph}(cn) \) is symmetric, i.e., if \( l(u, q, u') \in cn \) then \( l(u', q', u) \in cn \) for some \( q' \).

6. Connectedness: \( \text{graph}(cn) \) is connected, i.e., if \( n(u, t), n(u', t') \in cn \) then there is a path in \( \text{graph}(cn) \) connecting vertices \( u \) and \( u' \).

Condition 1 is essential for naming. Condition 2 is important, as we focus on closed systems. Condition 3 simplifies communication but could be lifted in principle. Conditions 4 and 5 are non-essential, but helpful. Finally, Condition 6 is essential for routing to stabilize, but many of the results below can be proved without it.

Objects and Tasks  In the type 2 semantics \textit{object containers} are now attached to a node \( u \) and have the shape

\[
o(o, a, u, q_{in}, q_{out})
\]

where \( o \in OID \), \( a \in OEnv \) as before, and \( q_{in}, q_{out} \) is a pair of an ingoing and an outgoing fifo message queue. This object level buffering is not essential, as messages are already buffered at link level, but object level buffering allows a more elegant formalization. It is commonplace in actor languages to consider inbound queues only. Here we find it more elegant to allow an outgoing queue as well, although this is mainly a matter of taste. Tasks \( t(o, l, s) \) are unchanged from the type 1 semantics.

Object Closures  Type 2 configurations are built from the four container types introduced above, nodes, links, objects and tasks. It remains to explain object closures. For an object message \texttt{OBJECT}(cn) to be valid, the configuration \( cn \) needs to be an \textit{object closure} of the form

\[
o(o, a, u, q_{in}, q_{out}) \ t(o, l_1, s_1) \ldots t(o, l_n, s_n)
\]

Specifically, if \( cn \) is any configuration then \( \text{clo}(cn, o) \), the \textit{closure} of object \( o \) with respect to \( cn \), is the multiset of all type 2 containers of the form either \( o(o', a', u', q'_{in}, q'_{out}) \) or \( t(o', l', s') \) such that \( o' = o \), and \( \text{objof}(cn) \) is a partial function returning \( o \) if all type 2 containers in \( cn \) are either objects or tasks, with OID \( o \).

Type 2 Runtime Syntax  Reflecting the above description, the type 2 runtime syntax is presented in Figure 2.7. A pictorial representation of the type 2 runtime state is shown in Figure 2.8. Configurations remain multisets, and we write, e.g., \( \text{obj} \in cn \) if \( cn \) can be written as \( \text{obj} \ cn' \) for some \( cn' \). Tasks are unchanged from Figure 2.4. We write \( t(cn) \) for the multiset of tasks in \( cn \), i.e., the multiset \( \{ tsk \mid \exists cn'. cn = tsk \ cn' \} \), and \( o(cn) \) for the multiset of objects in \( cn \), similarly defined. We use \( \preceq \) for the subterm relation, and write \( m(cn) \) for the multiset \( \{ msg \mid msg \preceq cn \} \). To avoid explosion of the notation we reuse symbols from the type 1 semantics as far as possible, and resolve them by context.
\[ u \in \text{NID} \]
\[ t \in \text{RTable} \quad = \quad \text{OID} \rightarrow \text{NID} \times \omega \]
\[ q \in Q \quad = \quad \text{Msg}^* \]
\[ \text{obj} \in \text{Obj}_2 \quad ::= \quad o(o, o', u, q_{in}, q_{out}) \]
\[ nd \in \text{Nd} \quad ::= \quad n(u, t) \]
\[ \text{lnk} \in \text{Lnk} \quad ::= \quad l(u, q, u') \]
\[ \text{ct} \in \text{ Ct}_2 \quad ::= \quad \text{tsk} \mid \text{obj} \mid nd \mid \text{lnk} \]
\[ \text{cn} \in \text{ Cn}_2 \quad ::= \quad ct_1 \ldots ct_n \]
\[ msg \in \text{Msg} \quad ::= \quad \text{call}(o, o', f, m, v) \mid \text{table}(t) \mid \text{OBJECT}(cn) \]

Figure 2.7: Type 2 runtime syntax

Figure 2.8: \textit{microABS-NET} runtime state

**Type 2 Reductions** An important distinction between the standard semantics and the network semantics is the absence of binding. For the standard semantics, name binding plays a key role to avoid clashes between locally generated names. However, in a language with NIDs this device is no longer needed, as globally unique names can be guaranteed easily by augmenting names with their generating NID. Since all name generation takes place in the context of a given NID, we can simply assume an operation \( \text{newo}(u) \) that returns a new OID, which is globally fresh for the "current configuration". Another important point to note is that all transitions in the type 2 semantics are \textit{fully local}, in the sense that all operations applied, and all conditions determining whether or not a transition is enabled, can be fully determined by inspecting only one node and, possibly, the head of incoming link queues, alternatively by enqueuing messages to the tail of the outgoing queue.

A number of reduction rules, for instance for most of the program constructs, are common to the type 1 and type 2 semantics, and deferred to [24]. The most interesting reduction rules are presented in Figure 2.9. The rules are naturally divided into subgroups:

- The rules \( \text{T-SEND} \) and \( \text{T-RCV} \) are concerned with the exchange of routing tables.
- The three rules \( \text{MSG-SEND}, \text{MSG-RCV} \) and \( \text{MSG-ROUTE} \) are used to manage message passing, i.e., reading a message from a link queue and transferring it to the appropriate object in-queue, and dually, reading a message from an out-queue and transferring it to the attached link queue. Finally, messages are routed to the next link, if the destination object does not reside at the current node. In rule \( \text{MSG-RCV} \)
The rules MSG-DELAY-1 and MSG-DELAY-2 we use the notation \( \text{nxt}(o, t) \uparrow \) to denote the condition that \( \text{nxt}(o, t) \) is undefined. These rules are used to handle the case where routing tables have not yet stabilized. For instance it may happen that updates to the routing tables have not yet caught up with object migration. In this case, a message may enter an out-queue without the hosting node’s routing table having information about the message’s destination (rule MSG-DELAY-2). Another case is where a node receives a message on a link without knowing where to forward it (rule MSG-DELAY-1). This situation is particularly problematic as a blocked message may prevent routing table updates to reach the hosting node, thus causing deadlock. The solution we propose is to use the network self-loop as a buffer for temporarily unroutable messages.

- The rules CALL-SEND and CALL-RCV produce and consume call messages in a pretty obvious way.
- The rule NEW-2 handles object creation, including registration of the new object at the local node.
- The final two rules concern object migration. Of these, OBJ-SEND is a global rule in that it is not allowed to be used in subsequent applications of the CTXT-1 rule. In this way we can guarantee that only complete object closures are migrated. In rule OBJ-SEND, \( cn - cn' \) is multiset difference.

We emphasize again that all of the above rules are strictly local and appeal only to mechanisms directly implementable at link level: Tests and simple datatype manipulations take place at a single node, or accesses
the nodes link layer interface. The “global” property appealed to above for the migration rules is merely a formal device to enable an elegant treatment of object closures.

The reduction rules can be optimized in several ways. For instance, object self-calls are always routed through the ‘network interface’, i.e., the hosting node’s self-loop. This is not necessary. It would be possible to add a rule to directly spawn a handling task from a self call without affecting the results.

In [24] we introduce a notion of well-formedness at type 2 level, which we leave out of this presentation.

### Type 2 Barbed Equivalence

We next adapt the notion of barbed equivalence to the type 2 setting. The only difficulty is to define the type 2 correlate of the observation predicate. We take the point of view that an observation \( \text{obs} = o!m(v) \) is enabled at a configuration \( cn \) if a corresponding call message \( \text{call}(o', o, m, v) \) is located at the head of one of the object output queues in \( cn \). More precisely, the type 2 observability predicate is \( cn \downarrow \text{obs} \), holding if and only if \( cn \) has the following shape;

\[
\begin{align*}
  cn &= cn' \circ(o', a, u, q_{in}, q_{out}) \\
\end{align*}
\]

and \( \text{hd}(q_{out}) \) is defined and equal to \( \text{call}(o', o, m, v) \).

There are other ways of defining the observability predicate that may be more natural. For instance one may attach external OIDs to specific NIDs and restrict observations to those NIDs accordingly. It is also possible to add dedicated output channels to the model, and route external calls to those. None of these design choices have any effect on the subsequent results, however, but add significant notational overhead, particular in the latter case.

With the observation predicate set up, the weak observation predicate is derived as in Section 2.2, and, as there, we define a type 2 witness relation as a relation that satisfies symmetry, reduction closure, and barb preservation. Thus:

**Definition 2.3.1** (Type 2 Barbed Equivalence). Let \( cn_1 \simeq_2 cn_2 \) if and only if \( cn_1 \mathcal{R} cn_2 \) for some type 2 witness relation \( \mathcal{R} \).

In fact, for our purpose there in no real need to distinguish between the type 1 and type 2 equivalences, and hence we conflate the notions of witness relation and barbed equivalences, by letting the type of the configuration arguments be determined by the context, and use \( \simeq \) as the generic notion.

### 2.4 Correctness

The goal is to prove that if \( cn_1 \) and \( cn_2 \) are initial type 1 and type 2 configurations, respectively, for the same program, then \( cn_1 \simeq cn_2 \). The key to the proof is a normal form lemma for the type 2 semantics saying, roughly, that any well-formed type 2 configuration can be rewritten, using a subset of the rules as detailed below, into a form where queues have been emptied of all routable messages, where routing tables have been in some expected sense normalized, and where all objects have been moved to a single node. We prove this in two steps. First we prove a stabilization result, that non-self links can be emptied of messages and routing tables normalized to induce messaging paths with unit stretch. This allows the second normalization step to empty also object queues and migrate all objects to a single node. Once this is done we can prove correctness by exhibiting a map representing each type 1 configuration as a canonical type 2 configuration, using normalization to help prove reduction preservation in both directions. Then only barb preservation is needed to complete the correctness argument.

**Stabilization** We first show that each configuration can be rewritten using the transition rules into a form for which routing is stable and all queues are empty, except for external messages, i.e., messages \( \text{msg} \) addressed to an object \( o \in \text{Ext} \). By well-formedness we then know that no object \( o(o', a', u', q_{in}, q_{out}) \preceq cn \) with \( o' = o \) exists. In the context of a configuration \( cn \) call a proper link any link \( l(u, q, u') \) for which \( u \neq u' \).

**Definition 2.4.1** (Stable Routing, External Link Messages). Let \( cn \) be a well-formed type 2 configuration.
Algorithm 1: Stabilize routing and read internal link messages

**Input** Type 2 well-formed configuration $cn$ on a connected network graph

**Output** Configuration with stable routing and external link messages only

repeat
  Use $T$-SEND on each proper link in $cn$ to broadcast routing tables to all neighbours;
  repeat
    Use $T$-RCV to dequeue one message on a link in $cn$ until $T$-RCV no longer enabled;
    Use $MSG$-RCV, $MSG$-ROUTE, $MSG$-DELAY-1, OBJ-RCV to dequeue one message from each link, if possible
  until link queues contain only external messages, and routing is stable

Figure 2.10: Algorithm 1 – Stabilize routing and empty link queues of internal messages

1. $cn$ has **stable routing**, if for all $(u, t), o(a, a', u', q_{in}, q_{out}) \leq cn$, if $nxt(o, t) = u''$ then there is a minimal length path from $u$ to $u'$ which visits $u''$.

2. $cn$ has **external link messages only**, if $l(u, q, u') \in cn$ and $msg \leq q$ implies $u = u'$ and $msg$ is external.

The strategy for performing the rewriting is to first empty link queues as far as possible as we simultaneously exchange routing tables to converge to a configuration with stable routing. This first stage is accomplished using Algorithm 1 in Figure 2.10 where we hide uses of $CTXT$-1 to allow the transition rules to be applied to arbitrary containers. Observe that we have no intention to use Algorithm 1 or any of the later algorithms in this section to do actual computing in the type 2 semantics. “Real” network computing using the type 2 semantics requires more sophisticated approaches. The algorithms considered here do not need to be effective or “local”: We only need to exhibit some strategy for producing a configuration with the desired result, allowing us to prove the desired normal form results.

**Proposition 2.4.2.** Algorithm 1 terminates.

Write $A_1(cn) \leadsto cn'$ if the configuration $cn'$ is a possible result of applying Algorithm 1 to $cn$. We then say that $cn'$ is in **stable form**. Stable forms are almost unique, but not quite, since routing may stabilize in different ways, and since this (plus the generally nondeterministic scheduling of rules in Algorithm 1) may cause messages to enter object input queues at different times. The main result, shown in [24] is that stabilization preserves barbed equivalence.

**Theorem 2.4.3.** If $A_1(cn) \leadsto cn'$ then $cn \simeq cn'$.

**Normalization** We then turn to the second normalization step, to empty object queues and migrate all object closures to a central node. The normalization procedure is Algorithm 2 shown in Figure 2.11. Let $A_2(cn) \leadsto cn'$ if $cn'$ is a possible result of applying Algorithm 2 to $cn$. Initially a node $u_0$ is chosen towards which all objects will migrate during normalization. Normalization is performed in cycles, each cycle starting and ending in a stable configuration. In each cycle, first object in- and out-queues are emptied. Then, objects not yet at $u_0$ are migrated one step towards $u_0$. Routing is not needed for this. It is sufficient to know that migration toward $u_0$ is possible.

**Proposition 2.4.4.** Algorithm 2 terminates.

Through a somewhat elaborate normal form argument we can establish the following result:

**Theorem 2.4.5.** If $A_2(cn) \leadsto cn'$ then $cn \simeq cn'$.
Algorithm 2: Normalization

Input Type 2 well-formed configuration \( cn \) on a connected network graph

Output Configuration in type 2 normal form

1. fix a NID \( u \);
2. run Algorithm 1;
3. repeat
   1. while some object queue is nonempty {
      1.1. use MSG-SEND, MSG-DELAY-2, CALL-RCV to dequeue one message from each nonempty object queue ;
   2. while an object exists not located at \( u \) {
      2.1. use OBJ-SEND to send the object towards \( u \) ;
   3. run Algorithm 1
4. until all objects are located at \( u \) and queues contain only external messages

Figure 2.11: Algorithm 2 – Normalization

Correctness The goal is to prove soundness and full abstraction of the network semantics, i.e., that for any two type 1 configurations \( \text{bind } o.cn, \text{bind } o'.cn' \) in standard form, \( \text{bind } o.cn \simeq \text{bind } o'.cn' \) if and only if \( \text{down}(cn_1) \simeq \text{down}(cn_2) \). However, since we have set up the semantics such that \( \simeq \) applies without modification at both type 1 and type 2 levels it suffices to prove that \( \text{bind } o.cn \simeq \text{down}(cn) \).

To accomplish this we represent each type 1 configuration as a type 2 configuration in normal form. We first fix an underlying graph represented as a well-formed type 2 configuration \( cn_{\text{graph}} \) and a distinguished UID \( v_0 \) in this graph, similar to the way initial configurations are defined in Section 2.3. Thus, \( cn_{\text{graph}} \) consists of nodes and links only, each node \( u \) in \( cn_{\text{graph}} \) has the form \( (u, t) \), and each link has the form \( (u, \varepsilon, u') \). The routing tables \( t \) are defined later. Defining a suitable representation map is a little cumbersome. A first complication is that names in the type 1 semantics (which includes the binder) need to be related to names in the type 2 semantics, which does not include the binder, but on the other hand has different generator functions (the function newo). For external names this is not a problem, but for bound names some form of name representation map is useful to connect the two types of names. Accordingly, we fix an injective name representation map \( \text{rep} \), taking names \( o \) in the type 1 semantics to names \( \text{rep}(o) \) in the type 2 semantics. For convenience we extend the name representation map \( \text{rep} \) to external names \( o \in \text{Ext} \) by \( \text{rep}(o) = o \), to arbitrary values by \( \text{rep}(p) = p \), to task environments by \( \text{rep}(l)(x) = \text{rep}(l(x)) \) and similarly for object environments. The only slight complication in defining the mapping \( \text{down} \) is that we need an operation to send a type 1 call container as a message in the type 2 semantics. This is done by the operation \( \text{send} \) which sends a call container originating at \( o \) onto object \( o' \)'s output queue as follows:

\[
\text{send}(c(o, o', m, v), o(a, a, u, q_{in}, q_{out}) \ cn) = o(a, a, u, q_{in}, \text{enq}(\text{CALL}(o, o', m, v), q_{out})) \ cn
\]  

where \( \text{enq}(\text{CALL}(o, o', m, v), q_{out}) \) is defined by induction on the structure of \( cn \) as follows:

- \( \text{down}(0)(cn) = cn \).
- \( \text{down}(cn_1 \ cn_2) = \text{down}(cn_1) \circ \text{down}(cn_2) \).
- \( \text{down}(t(o, l, s))(cn) = t(\text{rep}(o), \text{rep}(l), s) \ cn \).
- \( \text{down}(o(o, a))(cn) = o(\text{rep}(o), \text{rep}(a), u_0, \varepsilon, \varepsilon) \ cn \).
- \( \text{down}(c(o, o', m, v))(cn) = \text{send}(c(\text{rep}(o), \text{rep}(o'), m, \text{rep}(v)), cn) \).
In other words, we represent type 1 configurations by first assuming some underlying network graph, and then mapping the containers individually to type 2 level. The only detail remaining to be fixed above is the routing tables. For \( u_0 \) the initial routing table, \( t_0 \), needs to register all objects in \( cn_0 \), i.e.,

\[
t_0 = \text{reg}(g(o_0), u_0, \text{reg}(g(o_1), u_0, \text{reg}(\cdots, \text{reg}(g(o_m), u_0, \perp)) \cdots))
\]

where \( o_0, \ldots, o_m \) are the OIDs in \( cn_0 \). For nodes \( n(u, t) \) where \( u \neq u_0 \) we let \( t \) be determined by some stable routing. This is easily computed using Algorithm 1, and we leave out the details. This completes the definition of \( \text{down}(cn) \).

Lemma 2.4.6. Let \( \text{bind} \ z.cn \) be type 1 well-formed in standard form.

1. If \( \text{bind} \ z.cn \rightarrow \text{bind} \ z'.cn' \) then \( \text{down}(cn) \rightarrow^* o \simeq \text{down}(cn') \).

2. If \( \text{down}(cn) \rightarrow cn'' \) then for some \( z', cn', \text{bind} \ z.cn \rightarrow^* \text{bind} \ z'.cn' \) and \( cn'' \simeq \text{down}(cn') \). \( \square \).

We can now prove the main correctness result.

Theorem 2.4.7 (microABS-NET Implementation Correctness). For all well-formed type 1 configurations \( cn \) on connected network graphs, \( cn \simeq \text{down}(cn) \). \( \square \)

2.5 Discussion

We have presented a sound and fully abstract semantics for a rudimentary object language, in terms of a novel network-based execution model. Thanks in part to the explicit mixing of messaging and routing we are able to present the model at a level where it could in principle be implemented in a provably correct fashion directly on top of silicon, or integrated in a hypervisor such as Xen [9], assuming reliable link layer functionality only. This is a direction of work we aim to explore in the future.

Soundness and full abstraction is a useful validation that the network semantics induces the same behaviour on microABS programs as the reference semantics. The network semantics, however, lacks a scheduler to determine, e.g., when to migrate objects and how to schedule threads on single nodes. Such a scheduler will resolve nondeterministic choices left open in the network semantics presented here. Once such a scheduler is added, soundness and full abstraction is lost. As we show in the following chapter we can, however, easily adapt the results to a notion of barbed simulation, obtained by instead of requiring preservation of observations and reductions in both directions, requiring preservation only in one. Then correctness for barbed simulation, that \( \text{down}(cn) \) simulates \( \text{bind} \ o.cn \), is obtained as a corollary.

Substantial work has been going on in the HATS project on the ABS language [56] and its extensions, for instance towards software product lines [76]. Johnsen et al. [59] suggest an extension of ABS with deployment components for resource management. We are mainly interested in the microABS language as an example. Essentially, however, our work is language independent, and we could apply the approach presented here to a version of core Erlang with minor changes only. Some details would be different, in particular the treatment of Erlang’s pattern match-based message reception construct. The changes, however, would be local only, and so make little essential difference.

Much work has been done on object/component mobility in the \( \pi \)-calculus tradition [66], and on the implementation of high-level object or process-oriented languages in terms of more efficiently implementable low level calculi. In [78], following earlier work on Pict [70], Fournet’s distributed join-calculus [41], and the JoCaml programming language [22], a compiler is implemented and proved correct for Nomadic Pict, a prototype language with very similar functionality to our microABS language: principally asynchronous message passing between named, location-oblivious processes. The target language extends Pierce and Turner’s Pict language with synchronous local communication and asynchronous message passing between located processes. In comparison with [78] the use of name-based routing allows us to use barbed equivalence in place of coupled simulation [69] and as a consequence obtain a simpler correctness proof, due to the need for locking in the central forwarding server scheme used in [78]. JoCaml also uses forward chaining, along with an
elaborate mechanism to collapse the forwarding chains. In the Klaim project [12] compilers were implemented and proved correct for several variants of the Klaim language, using the Linda tuple space communication model and a centralized name server to identify local tuple servers. The Oz kernel language [81] uses a monotone shared constraint store in the style of concurrent constraint programming. The Oz/K language [63] adds to this a notion of locality with separate failure and mobility semantics, but no real distribution or communication semantics is given (long distance communication is reduced to explicit manipulation of located agents, in the style of Ambient calculus [16]).

Our correctness proof uses reduction semantics and barbed equivalence. This is rather standard in the process algebra literature, cf. [40, 19, 43]. Both Sewell et al. [78] and Fournet et al. [42] use coupled simulation in order to handle problems related to preemptive choice. This complication does not arise in our work, whence barbed equivalence suffices. However, barbed equivalence is mostly useful for closed systems modeling where the stimuli to which an observed system is to be exposed must be given up front, as part of the initial configuration. A structural account based on some form of contextual equivalence [71], or on bisimulation equivalence along with a labelled transition semantics instead, would be more suitable. Work in this direction is currently going on. Replacing our reference reduction-based semantics with a labelled transition semantics is fairly straightforward. The bigger challenge is to develop a suitable structural account at the network level, allowing partially defined configurations to be composed.

Another direction for future work is to extend the microABS language. Experiments in this directions are going on. In [26], reported in the next chapter, we extend microABS with futures (aka promises [64]) as placeholders for return values. This extension turns out to be not at all trivial. In other directions it is of interest to consider models with node power-on/power-off, in order to model systems with adaptive power consumption, as well as various forms of node failure, along similar lines as [37]. The model can be used as a platform for language-based studies of load balancing and resource adaptation. We have extended the network semantics reported here to the full core ABS language [56] and implemented a multi-core simulation engine. This work is reported in [25]. There we show how the network semantics presented here can be split into a language interpreter layer and a language independent node controller layer, not unlike the meta-actors of [62], and we show how different resource allocation heuristics can be used to optimize object-to-node placement for different simple applications.

Further down the line it is of interest to examine the potential practical implications of our approach. This presents additional challenges, including garbage collection and buffer management, in particular as the network semantics we have presented above uses unbounded buffers. Also the routing scheme must be reconsidered. Distance vector routing suffers from well-known and fundamental scalability and security problems, and needs consideration in light of recent progress on compact routing (cf. [83, 79]).
Chapter 3

Efficient Routing of Futures in Object Networks

3.1 Introduction

In [24], summarized in Chapter 2 of this deliverable, we introduce a novel model for highly adaptable and efficient networked execution of object-based programs. The key idea is to use a form of location independent, or flat, routing [51, 14, 53] that allows messages (RPCs) to be routed directly to the called object, independent of the physical node on which that object is currently executing. In this way a lot of the overhead and performance constraints associated with object mobility can be eliminated, including latency and bandwidth overhead due to looking up, querying, updating, and locking object location databases, and overhead due to increased traffic, for instance for message forwarding.

The language microABS considered in Chapter 2 can be viewed as a candidate for a minimally functional object-based language. Essentially it allows to define a dynamically growing or shrinking collection of objects communicating by asynchronous RPC, and thus its functionality is not much different from a core version of Erlang [17], or the nomadic PICT language studied in [78]. The question is how program behaviour is affected by being run in the networked model, as compared with a more standard (reference) semantics given, for instance, using rewrite logic. This comparison is of interest, since the reference semantics is given at a high level of abstraction and ignores almost all aspects of physical distribution, such a location, routing, message passing, and so on. In Chapter 2 we show that, with a maximally nondeterministic networked semantics, and in the sense of barbed equivalence [73] which is a standard equivalence to study in these types of applications [19, 40, 43], programs exhibit the same behaviour in both cases.

Messaging in the microABS setting is very simple. The implicit channel abstraction used in the reference semantics for microABS in Chapter 2, and for the various ABS dialects in general, is essentially that of a reliable, unordered communication channel. Messages (calls) are transmitted according to the program order. Methods are received synchronously at any time, but in the fully asynchronous model considered here scheduling does not introduce any happens-before constraints on observable events arising from racing (i.e., simultaneously enabled) method calls. Soundness and full abstraction for the networked semantics is therefore an interesting and useful observation, since it allows many conclusions made at the level of abstract program behaviour to be transferred to the setting of a networked realization.

The question is how sensitive this observation is to the type of communication taking place at the abstract level. For this reason it is of interest to examine also languages with richer communication structures than asynchronous point to point message passing. To this end we enrich in this chapter the microABS language studied earlier with future variables and show that the conclusions of our previous work remain valid, however

---

1 This is strictly speaking not true in general, as in the reference semantics, the program order on calls may induce happens-before constraints on external method calls that cannot be realized in the networked semantics, because messages are explicitly queued and can always be shuffled. However, barbed equivalence is not sensitive to this type of happens-before constraint in the reference semantics.

25
with more involved constructions.

Futures \([18, 39, 64, 87, 68, 29]\) are placeholder variables for values that may be waiting to get instantiated. Futures are used extensively in concurrent and distributed high-level languages, libraries, and models including Java, .NET, Scheme, concurrent LISP, Oz, to name just a few. Our work relies on the future model introduced in \([29]\) where futures are introduced as placeholders for return values of remote method calls, the model also underpinning the use of futures in ABS generally. Other models exist, such as the transparent futures considered in \([15]\), or the concurrent constraint store model of Oz \([81]\). ABS has other constructs of relevance, such as the notion of concurrent object group (COG, \([76]\)), but this is introduced in ABS mainly as a container for local evaluation, to support, within the context of a single computational unit, the collaborative scheduling model also introduced in \([29]\). Since our focus here is squarely on large scale message passing concurrency, it is therefore not too farfetched to claim that the language and model of asynchronous object networks with futures studied here is a reasonable candidate for a networked ABS core language.

The contribution reported in \([26]\) and summarized in the present chapter, then, is an extension of the networked semantics of Chapter 2 to a richer fragment of ABS including futures, and a correctness proof showing that soundness and full abstraction remains valid in this richer setting.

The chapter is organized in the following way: In Section 3.2 we briefly introduce the extended version of microABS, called milliABS, and sketch its reduction semantics, as in Chapter 2. For details we refer the reader to the full version \([26]\). In Section 3.4 we present the network semantics, runtime states and the reduction relation, and in Section 3.5 we sketch the correctness (soundness and full abstraction) proof. We conclude the chapter by briefly discussing the implications of scheduling on the correctness analysis. Proofs are deferred to the full version \([26]\).

### 3.2 mABS

We define a small concurrent object-based language milliABS with asynchronous calls and futures, as depicted in Figure 3.1. The milliABS language is an extension of the language microABS of message-passing processes introduced in the previous chapter with return values as futures. The addition of futures is the main difference in relation to microABS. The idea is that a method call, say, `foo!bar(x)` returns immediately with an uninitialized future `f`, as a placeholder for a not-yet-defined return value, at the same time spawning an invocation of method `bar` on object `foo`, `foo.bar(x)`, that will at some later time cause the return value to become initialized. A task can at any time attempt to retrieve the value of a future `f`, by executing `f.get`. If `f` is not yet instantiated this will cause the calling task to be suspended. The main benefit of futures is that long-running computations can be spawned off to execute in parallel without holding up the caller until the return value is actually needed. Futures are central to this work. One of our main objectives, above and beyond \([24]\), is to show how futures can be implemented in a sound and fully abstract fashion on a large scale network in a fully decentralized and network transparent fashion.

**Example 3.2.1.** A very simple server applying `foo` to its argument is shown in Figure 3.2.
Example 3.2.2. Assume that $\text{combine}(\text{upper}(x), \text{lower}(x)) = \text{foo}(x)$. The program example in Figure 3.3 returns immediately with the result, if the argument to serve is small. If the argument is not small, two new servers are spawned, and computation of the result on upper and lower tranches is delegated to those servers. The results are then fetched from the two newly spawned servers by evaluating the get statements, combined, and returned.

3.3 Reduction Semantics

We first present an abstract “reference” semantics for milliABS based on rewriting logic. The presentation follows Chapter 2 quite closely. We use the abstract semantics as the point of reference for the concrete network-oriented semantics which we present later. The goal is to show that the concrete network semantics correctly implements the abstract semantics in the sense of barbed equivalence. The reduction semantics uses a reduction relation $cn \rightarrow cn'$ where $cn, cn'$ are configurations, as determined by the runtime syntax in Figure 3.4. Later on, we introduce different configurations and transition relations, and so use index 1, or talk of e.g., configurations of “type 1”, for this first semantics when we need to disambiguate. We highlight only those aspects of the runtime syntax that are changed with respect to Chapter 2. The runtime syntax now has four types of containers: Tasks, objects, futures, and calls. Tasks and calls are essentially unchanged from the microABS semantics. Futures are used as centralized stores for assignments to future variables. Task and object environments $l$ and $a$, respectively, map local variables to values. Task environments are aware of a special variable $\text{ret}$ that the task can use in order to identify its return future. Upon invocation, the task environment is initialized using the operation $\text{locals}(o, f, m, v)$ by mapping the formal parameters of $m$ in $o$ to the corresponding actual parameters in $v$, by initializing the method local variables to suitable null values, by mapping $\text{self}$ to $o$, and by mapping $\text{ret}$ to $f$, intended as the return future of the task being created. Object environments are initialized using the operation $\text{init}(C, v)$, which maps the parameters of $C$ to $v$, and initializes the object local variables as above.

The reduction rules of Figure 2.5 are unchanged for milliABS. The remaining rules in 3.5, replacing those of Figure 2.6, address the more interesting cases that involve inter-object communication, external method
invocation, and object creation. A method call causes a new future to be created, along with its future. We now turn to the second part of the chapter where we address the problem of efficiently executing \textit{milliABS} programs on an abstract network graph using the name-based routing scheme introduced in Chapter 2. A commonplace approach to implementation of futures is by forward chaining. In \textit{ABS}, futures are produced as placeholders for return values. Each object mentioning a future can subscribe to that future at some other object. This may happen in remote method calls where the caller subscribes to the return value later to be provided by the callee. It may also happen when a value containing a future is passed from some sender object to some receiver object. In that case the receiver object becomes subscriber at the sender object for that future. When a future gets instantiated to an actual value at some object, it is the task of that object to forward the instantiation to the subscribing objects. This is the implementation strategy applied in our work as well, and it is the objective of the proof to show that this approach is sound and fully abstract for our network semantics, even when routing is in an unstable state.

They are very meaningful in a labelled semantics setting, but that is a different story.
Other language models and implementation strategies exist, besides the one adopted here. It is, for instance, possible to lift futures to become first class objects with an explicit instantiation method. This design choice introduces the possibility of race conditions. In the ABS language this complication is avoided, as future instantiations are always rooted in some return statement, a property we rely on heavily in the technical development below. It is also possible to adopt a lazy strategy for instance propagation, instead of the eager strategy adopted here, where future instantiations are forwarded as soon as they become available, regardless of whether the subscriber of the instantiation actually has a need for the value. A possible alternative design is to instead request an instance when needed. This has the disadvantage of introducing additional delays at runtime, something which may nullify, to some extent, the rationale of introducing futures in the first place. This is discussed in more detail in the following chapter.

### 3.4.1 milliABS-NET Runtime Syntax

In Figure 3.6 we present the milliABS-NET runtime syntax, i.e., the shape of the runtime state. Recall from Section 3.3 that we reuse symbols as much as possible and use indices to disambiguate. Thus, for instance, $\text{Obj}_1$ is the set $\text{Obj}$ of the type 1 semantics in Figure 3.4, and $\text{Obj}_2$ is the corresponding set in Figure 3.6. We adopt the same syntactical conventions as in Section 3.3. Tasks are unchanged from Figure 3.4. We write $t(cn)$ for the multiset of tasks in $cn$, i.e., the multiset $\{tsk \mid \exists cn'. cn = tsk cn'\}$, and $o(cn)$ for the multiset of objects in $cn$, similarly defined. We also write $m(cn)$ for the multiset $\{msg \mid msg \preceq cn\}$.

We proceed to explain the different types of containers and the operations on them, highlighting the differences re. the treatment in Chapter 2. For a detailed explanation of other features such as routing, message passing, and the network graph, we refer to the previous chapter, and to [24][26].

**Objects and Object Environments** Objects $o(a, a, u, q_{in}, q_{out})$ are now attached to a node $u$ and a pair of an incoming ($q_{in}$) and an outgoing ($q_{out}$) fifo message queue, and the notion of object environment is refined to take futures into account in a localized manner. In the type 2 semantics, object environments $a$ are now augmented by mapping futures $fut$ to pairs $(v, o)$ where:

- $v$ is the lifted value currently assigned to $fut$ at the current object, and
- $o$ is a forwarding set of the objects subscribing to updates to $fut$ at the current object.

For instance, if $a(fut) = (\bot, o_1 :: o_2 :: \varepsilon)$ the future $fut$ is as yet uninstantiated (at the object to which $a$ belongs), and, if $fut$ eventually does become instantiated, the instantiation must be forwarded to $o_1$ and $o_2$, in random order.

We introduce some syntax to help manipulating object environments:

- $a(x)$ abbreviates $\pi_1(a)(x)$, $a(f)$ abbreviates $\pi_2(a)(f)$
\* \* a[v/x] is a with \( \pi_1(a) \) replaced by the expected update. Similarly \( a[v/f] \) updates \( \pi_2(a) \) by mapping \( f \) to the pair \((v, \pi_2(a(f)))\), i.e., the assigned value is updated and the forwarding list remains unchanged. If \( f \notin \text{dom}(\pi_2(a)) \) then \( a[v/f](f) = (v, \varepsilon) \), i.e., the update to value takes effect. Finally we use \( a[(v, o)/f] \) for the expected update where both the value and the forwarding list is updated.

\* \* fut\((v, o, a)\) updates \( \pi_2(a) \) by for each future \( f \) occurring in \( v \) adding \( o \) to the forwarding list of \( a(f) \), i.e., by mapping \( f \) to the pair either \((\perp, o)\) if \( a(f) \) is undefined \((= \perp)\), or \((\pi_1(a(f)), o :: \pi_2(a(f)))\) otherwise.

\* \* init\((C, v)\) returns an initial object environment by mapping the formal parameters of \( C \) to \( v \).

\* \* init\((f, a)\) augments \( a \) by mapping \( f \) to the pair \((\perp, \varepsilon)\). If \( f \notin \text{dom}(a) \) then \( \text{init}(f, a) = a \).

\* \* init\((v, a)\) augments \( a \) by mapping each \( f \) in \( v \) which is uninitialized in \( a \) (i.e., such that \( f \notin \text{dom}(a) \)) to \((\perp, \varepsilon)\).

As a consequence of this change, futures are eliminated as containers in the type 2 runtime syntax. In other respects, the type 2 runtime syntax is unchanged: Syntactical conventions that are not explicitly modified in the type 2 syntax above are unchanged, in particular we continue to assume multiset properties of configuration juxtaposition.

**Messages** In relation to Chapter 2 call messages are modified slightly, to the shape \( \text{CALL}(o, o', f, m, v) \) in order to record the identity of the associated future, and a future instantiation message \( \text{FUTURE}(o, f, v) \) with destination \( o \) is added. Both call messages and future messages are object bound. Other concepts such as object closures are unchanged from Chapter 2.

### 3.4.2 Reduction Semantics

As noted in Chapter 2, an important distinction between the standard semantics and the network semantics is the absence of binding. For the standard semantics, name binding plays an important role to avoid clashes between locally generated names. However, in a language with NIDs this device is no longer needed, as globally unique names can be guaranteed easily by augmenting names with their generating NID. Since all name generation in the milliABS-NET semantics below takes place in the context of a given NID, we can simply assume operations \( \text{newf}(u) \), resp., \( \text{newo}(u) \), that return a new future, resp., OID, which is globally fresh for the “current context”. We use \( \text{new}(z) \) for either \( \text{newf} \) or \( \text{newo} \) when the nature of \( z \) is not known.

We present the milliABS-NET reduction rules. First, figure 2.5 applies with the following two minor modifications:

- Rule CTXT-2 is dropped as name binding is dropped from the type 2 runtime syntax

- Rule WFIELD is modified in the obvious way to read: If \( x \in \text{dom}(a) \) then \( o(a, a, u, q_{in}, q_{out}) \ t(o, l, x = e; s) \rightarrow o(a, a[e(a, l)/x], u, q_{in}, q_{out}) \ t(o, l, s) \)

Next, the rules in Figure 2.6 also appear unchanged, with the exception of rules \text{CALL-SEND} and \text{CALL-RCV}. These are replaced by the six rules in Figure 3.7. The four rules \text{CALL-SEND}, \text{CALL-RCV}, \text{FUT-SEND}, \text{FUT-RCV} produce and consume messages, method calls and future instantiations. A method call causes a local future to be created and passed with the call message. Upon reception of the call, the callee first initializes those received futures it does not already know about, and then augments the resulting local object environment to forward instantiations of the received future to the caller. Observe that it may be that the callee already knows about the return future of the call. Since message order is not assumed to be preserved a later call referring to the original return future may overtake the earlier call. The eventual return value becomes bound to the return future by the assignment to the constant \text{ret} during initialization of the called methods local environment. The rule \text{FUT-SEND} may cause future instantiations to be forwarded to objects in the forwarding list whenever the future is seen to have received a value, and \text{FUT-RCV} causes the receiving object
CALL-SEND: Let \( o' = e_1(a,l), v = e_2(a,l) \), \( f = \text{newf}(u) \) in
\[
\begin{align*}
&\sigma(o,a,u,q_{in},q_{out}) \quad t(o,l,x = e_1\text{in}(e_2);s) \rightarrow \\
&\sigma(o,fw(v,o',\text{init}(f,a)),u,q_{in},\text{enq}(\text{CALL}(o,o',f,m,v),q_{out})) \quad t(o,l[f/x],s)
\end{align*}
\]

CALL-RCV: If \( hd(q_{in}) = \text{CALL}(o',o,f,m,v) \) then \( o(o,a,u,q_{in},q_{out}) \rightarrow \\
\sigma(o,fw(f,o',\text{init}(v,\text{init}(f,a))),u,\text{deq}(q_{in}),q_{out}) \\
\quad t(o,\text{locals}(o,m,f,v),\text{body}(o,m))
\]

FUT-SEND: If \( a(f) = (v,a_1::a_2) \) then
\[
\begin{align*}
&\sigma(o,a,u,q_{in},q_{out}) \rightarrow o(o,fw(v,o',a([v,o_2]/f]),u,q_{in},\text{enq}(\text{FUTURE}(o_1,f,v)),q_{out})
\end{align*}
\]

FUT-RCV: If \( hd(q_{in}) = \text{FUTURE}(o,f,v) \) then
\[
\begin{align*}
&\sigma(o,a,u,q_{in},q_{out}) \rightarrow o(o,a[v/f],u,\text{deq}(q_{in}),q_{out})
\end{align*}
\]

RET-2: \( o(o,a,u,q_{in},q_{out}) \quad t(o,l,\text{return } e;s) \rightarrow o(o,a[e(a,l)/l(\text{ret})],u,q_{in},q_{out})
\]

GET-2: If \( e(a,l) = f \) and \( a(f) = v \) then
\[
\begin{align*}
&\sigma(o,a,u,q_{in},q_{out}) \quad t(o,l,x = e\text{.get};s) \rightarrow t(o,l[v/x],s)
\end{align*}
\]

Figure 3.7: milliABS-NET reduction rules

to update its local environment accordingly. A future may itself be instantiated to a future. The local forwarding table may thus need to be updated.

The two rules RET-2 and GET-2 handle the corresponding language constructs. Return statements cause the corresponding future to be instantiated, as explained, and get statements read the value of the future provided it has received a value.

We leave again the definitions of initial configuration, reachability, and type 2 well-formedness to \[26\]. It must be noted that the well-formedness conditions become rather more subtle in the type 2 case, as various consistency properties of the network semantics must be verified, for instance to ensure that assignments to futures are always consistent. Whereas this is trivial in the type 1 case since futures there are uniquely represented, in the type 2 case a much more involved analysis is needed.

### 3.5 Normal Forms

An milliABS-NET program can be run from an initial state in either the type 1 or the type 2 semantics. We want to show that the behaviour of the programs is preserved, in the sense that the initial states at type 1 and type 2 are barbed equivalent, referring to Chapter 2 for the definition of barbed equivalence.

The key to the proof is a normal form lemma for milliABS-NET saying, roughly, that any well-formed type 2 configuration can be rewritten into a form where queues have been emptied of all routable messages, where routing tables have been in some expected sense normalized, where all futures that are assigned a value somewhere are assigned a value everywhere the value might be needed (by well-formedness this value is unique), and where all objects have been moved to a single node. We perform this rewriting in two steps:

- First we show that routing can be stabilized and inter node links emptied, except for external messages (messages addressed at an external OID). This part is unchanged from Chapter 2 and not repeated here.
- We then complete the construction by emptying object queues, propagating futures, and moving all objects to a single node. This is done in the following section.
Algorithm 3: Normalization

**Input** Well-formed type 2 configuration $cn$ on a connected network graph

**Output** Configuration in type 2 normal form

fix a NID $u$;
run Algorithm 1;
while some object queue is nonempty {
use MSG-SEND, MSG-DELAY-2, CALL-RCV, FUT-RCV to dequeue one
message from each nonempty object queue;
while FUT-SEND is enabled { apply FUT-SEND } };
while an object exists not located at $u$ {
use OBJ-SEND to send the object towards $u$;
run Algorithm 1 }

Figure 3.8: Algorithm 3 – Normalization

3.5.1 Normalization

The normalization procedure, Algorithm 3, shown in Figure 3.8 is a minor extension of alg. 2 of the previous section. Let $A_3(cn) \leadsto cn'$ if $cn'$ is a possible result of applying Algorithm 3 to $cn$. Initially a node $u$ is chosen towards which all objects will migrate during normalization. Normalization is performed in cycles, each cycle starting and ending in a stable configuration. In each cycle one message is read from the object in- and out-queues. By well-formedness, object queues contain only calls and future messages. Receptions of future messages may cause the object environment to instantiate futures. This may cause new future instantiation messages to be enabled. Accordingly, those messages are generated and sent to the objects out-queue. Once this is done, objects not yet at $u$ will be migrated.

**Proposition 3.5.1.** Algorithm 3 terminates

We then turn to normal forms and define first a couple of auxiliary operations. Let $t_2(cn)$ be the multiset of method containers $t_{sk} = t(o,l,s)$ such that one of the following cases apply:

- $t_{sk}$ is a task container in $cn$.
- There is a message $\text{CALL}(o',o,f,m,v)$ in transit, $o$ is not external, $l = \text{locals}(o,m,f,v)$ and $s = \text{body}(o,m)$.

Let $o_2(cn)$ be the multiset of object containers $o(o,a,u,\varepsilon,\varepsilon)$ for which the following apply:

- There is an object container $\text{obj} = o(o,a',u',q_{in},q_{out}) \preceq cn$
- $a'(x) = a(x)$ for all variables $x$
- $a'(f) = (v,\varepsilon)$ with $v \neq \bot$ if, and only if, for some object container $o(a_1,a_1,u_1,q_{in,1},q_{out,1}) \preceq cn$, $a_1(f) = (v,o)$

Also say that $cn$ has external messages only, if link queues in $cn$ contain only external messages.

**Definition 3.5.2** (Normal Form). A well-formed configuration $cn$ is in normal form, if

1. $cn$ has stable routing
2. $cn$ has external messages only
3. $t(cn) = t_2(cn)$
4. $o(cn) = o_2(cn)$

32
Proposition 3.5.3. Suppose \( cn \) is well-formed. If \( A_3(cn) \leadsto cn' \) then

1. \( cn' \) is in normal form
2. \( graph(cn) = graph(cn') \)
3. \( t_2(cn) = t(cn') \)
4. \( o_2(cn) = o(cn') \)
5. \( m_1(cn) = m(cn') \)

Proposition 3.5.3 motivates the following definition of normal form equivalence.

Definition 3.5.4 (\( \equiv \)).

1. Let \( cn_1 R_2 cn_2 \) if and only if \( cn_1 \) and \( cn_2 \) are both well-formed, \( graph(cn_1) = graph(cn_2) \), \( t_2(cn_1) = t_2(cn_2) \), \( o_2(cn_1) = o_2(cn_2) \), and \( m_1(cn_1) = m_1(cn_2) \).

2. Let \( cn_1 \equiv cn_2 \) if and only if there are \( cn'_1, cn'_2 \) such that

\[
A_3(cn_1) \leadsto cn'_1 R_2 cn'_2 \leadsto A_3(cn_2)
\]

We obtain that normalization respects normal form equivalence.

Corollary 3.5.5. If \( A_3(cn) \leadsto cn' \) then \( cn \equiv cn' \)

The following lemma is central to the use of normal forms.

Lemma 3.5.6. \( \equiv \) is reduction closed.

Proposition 3.5.7. If \( cn_1 \equiv cn_2 \) then \( cn_1 \simeq cn_2 \).

Corollary 3.5.8. If \( A_3(cn) \leadsto cn' \) then \( cn \simeq cn' \).

3.6 Correctness

In this section we prove correctness of the network semantics by mapping each well-formed type 1 configuration \( \text{bind } z.cn \) in standard form to a well-formed type 2 configuration \( \text{down}(cn) \) on an arbitrary underlying network graph. We then prove that the two configurations are barbed equivalent, i.e., that \( \text{bind } z.cn \simeq \text{down}(cn) \).

Defining the Underlying Network Graph We first fix an underlying graph represented as a well-formed type 2 configuration \( cn_{\text{graph}} \) with a distinguished UID \( u_0 \), similar to the way initial configurations are defined in Section 3.4. Thus, \( cn_{\text{graph}} \) consists of nodes and links only, each node \( u \) in \( cn_{\text{graph}} \) has the form \( (u, t) \), and each link has the form \( (u, \varepsilon, u') \). The routing tables \( t \) are defined later.
Representing Names and Values  To represent names, one complication is that names in the type 1 semantics need to be related to names in the type 2 semantics, which does not include the binding construct of the type 1 semantics, but on the other hand has different generator functions (the functions newf and newo). This prevents the name relation from being modeled using the identity relation. To address this we assume that names and futures in the type 1 semantics are really symbolic, connected to concrete names/futures used in the type 2 semantics by means of an injective name representation map rep, taking internal names f, o in the type 1 semantics to names rep(f), rep(o) in the type 2 semantics. For convenience we extend the name representation map rep to arbitrary values and task environments as follows:

- rep(0) = o, if o ∈ Ext,
- rep(p) = p, if p ∈ PVal,
- rep(l)(x) = rep(l(x)), rep(l)(ret) = rep(l(ret))

Representing Object Environments  To extend rep also to object environments, a complication is that object environments in the type 2 semantics must be defined partially in terms of the type 1 environments (for object variables) and partially in terms of the future containers available in the “root configuration”, since the type 1 semantics uses future containers in place of forwarding lists stored in object environments. To this end we first define an auxiliary function oenvmap(cn, p, rep) ∈ Fut → Val⊥ on triples of type 1 configurations, a pool of OID/future constants, and a name representation map, as a function which gathers together assignments to futures as determined by the future containers in cn:

- oenvmap(0, p, rep)(f) = oenvmap(tsk, p, rep) = oenvmap(obj, p, rep) = oenvmap(cl, p, rep) = ⊥
- oenvmap(f(f, v⊥), p, rep)(f′) = (if rep(f) = f′ then rep(v⊥) else ⊥)
- oenvmap(bind o.cn, p ∪ {o′}, rep)(f) = oenvmap(cn, p, rep[o′/o])(f)
- oenvmap(bind f.cn, p ∪ {f′}, rep)(f″) = oenvmap(cn, p, rep[f′/f])(f″)
- oenvmap(cn1 cn2, p, rep)(f) = oenvmap(cn1, p, rep)(f) ⊔ oenvmap(cn2, p, rep)(f)

Fix now a root type 1 configuration cn0 and a large enough pool p0 of names (proportional to the size of cn0, and computed using newf(u0) and newo(u0) to conform with our naming policy). Assume that cn0 is in standard form, i.e., cn0 = bind z0.cn′ 0 where cn′ 0 does not have binders. Fix g = oenvmap(cn0, p0, ⊥) and cngraph as above. We can now extend rep to object environments by:

- π1(rep(a))(x) = rep(π1(a))(x)
- π2(rep(a))(f) = \{(g(f), ε) if g(f) ≠ ⊥
- ( ⊥, OID(cn0)) otherwise

Representing Call Containers  Another complication is that we need an operation to represent a type 1 call container as a message in the type 2 semantics. This is done in the obvious way by the operation send as follows:

\[
\text{send}(c(o, o′, f, m, →), o(a, a, u, qin, qout) \ cn) = o(a, a, u, qin, \text{enq}(\text{CALL}(o, o′, f, m, →), gout)) \ cn
\] (3.1)
Representing Configurations  Given a name representation map $\text{rep}$ we can now define the representation of a type 1 configuration as a transformer on type 2 configurations, initially the underlying network graph, as the mapping $\text{down}$ as follows:

- $\text{down}(0, \text{rep})(cn) = cn$
- $\text{down}(cn_1, cn_2, \text{rep}) = \text{down}(cn_1, \text{rep}) \circ \text{down}(cn_2, \text{rep})$
- $\text{down}(t(o, l, s), \text{rep})(cn) = t(\text{rep}(o), \text{rep}(l), s) \circ cn$
- $\text{down}(o(o, a), \text{rep})(cn) = o(\text{rep}(o), \text{rep}(a), u_0, \varepsilon, \varepsilon) \circ cn$
- $\text{down}(f(f, \rightarrow \perp), \text{rep})(cn) = cn$
- $\text{down}(c(o, o', f, m, \rightarrow), \text{rep})(cn) = \text{send}(c(o, o', f, m, \rightarrow), cn, u_0)$

In other words, we represent type 1 configurations by first assuming some underlying network graph, secondly distributing the (centralized) assignments to futures in each object environment with the trivial forwarding lists, and then, once this is done, mapping the containers individually to type 2 level.

Defining Routing Tables  The only detail remaining to be fixed above is the routing tables. For $u_0$ the initial routing table, $t_0$, needs to register all objects in $cn_0$, i.e.,

$t_0 = \text{reg}(g(o_0), u_0, \text{reg}(g(o_1), u_0, \text{reg}(\ldots, \text{reg}(g(o_m), u_0, \bot)) \ldots))$

where $o_0, \ldots, o_m$ are the OIDs in $cn_0'$. For nodes $(u, t)$ where $u \neq u_0$ we let $t$ be determined by some stable routing. This is easily computed using Algorithm 1, and we leave out the details.

Definition 3.6.1 (Representation Map $\text{down}$). Let a network graph $cn_{\text{graph}}$ and a name representation map $\text{rep}$ be given. For each well-formed type 1 configuration $cn_0$, the type 2 representation of $cn_0$ is the configuration $\text{down}(cn_0) = \text{down}(cn_0, \text{rep})(cn_{\text{graph}})$.

In this definition, forwarding lists are overapproximated as compared to the type 2 semantics, which forward futures only to objects that have actually received them. To handle this slight complication we need a little lemma saying that for well-formed type 2 configurations, forwarding lists can be extended without affecting observable behaviour. To make this precise say that $\sigma(o, a, u, q_{in}, q_{out})$ extends $\sigma(o', a', u', q'_{in}, q'_{out})$, if $o = o'$, $u = u'$, $q_{in} = q'_{in}$, $q_{out} = q'_{out}$, $a(x) = a'(x)$ for all $x$, and $\pi_1(a(f)) = \pi_1(a'(f))$ and $\pi_2(a(f)) \supseteq \pi_2(a'(f))$ for all $f$.

Lemma 3.6.2. Suppose that $cn$ is type 2 well-formed, and $cn'$ differs from $cn$ only by replacing each object $\text{obj}$ by an object $\text{obj}'$ such that $\text{obj}'$ extends $\text{obj}$. Then $cn'$ is type 2 well-formed as well, and $cn \simeq cn'$.

As in Chapter 2 we can now show a key lemma allowing us to relate transitions in the two semantics under barbed equivalence.

Lemma 3.6.3. Let $\text{bind } z.cn$ be type 1 well-formed in standard form.

1. If $\text{bind } z.cn \rightarrow \text{bind } z'.cn'$ then $\text{down}(cn) \rightarrow^* o \simeq \text{down}(cn')$

2. If $\text{down}(cn) \rightarrow cn''$ then for some $z'$, $cn'$, $\text{bind } z.cn \rightarrow^* \text{bind } z'.cn'$ and $cn'' \simeq \text{down}(cn')$

We can now state the main result.

Theorem 3.6.4 (Correctness of the Type 2 Semantics). For all well-formed type 1 configurations $cn$ on a connected network graph,

$cn \simeq \text{down}(cn)$
3.7 Scheduling

The type 2 semantics is highly nondeterministic. The semantics says nothing about how frequently routing tables are to be exchanged, when messages should be passed between the different queues, when future messages are to be sent, and when, and to where, objects are to be transmitted. Resolving these choices is a crucial tradeoff between management overhead and performance. For instance, if routing tables are exchanged at a very high frequency, routing can be always assumed to be in stable state. This ensures short end to end routes, but at the expense of a large management overhead. This raises the question of how to determine these parameters, something which we address in more detail in the following chapter.

Regardless how this is done, a real implementation needs to resolve these choices. This is tantamount to eliminating nondeterminism from the type 2 semantics, essentially by removing transitions. Thus, in a sense, Theorem 3.6.4 achieves more than is called for, as soundness and full abstraction a priori applies only to the type 2 semantics with all transitions included.

A scheduler can be viewed abstractly as a predicate on histories in the following way. Let a scheduled execution be any sequence \( \epsilon = cn_0cn_1 \cdots cn_n \) such that \( cn_i \rightarrow cn_{i+1} \) for all \( 0 \leq i < n \) where the \( cn_i \) are well-formed type 2 configurations. A scheduler is a predicate \( S \) on such sequences, with the property that

1. \( S(\langle cn \rangle) \) for any \( cn \) where \( \langle cn \rangle \) is the one element execution consisting of \( cn \) (a scheduler kicks in only when execution is started), and

2. if \( S(cn_0 \cdots cn_n) \) and \( cn_n \rightarrow cn_{n+1} \) for some \( cn_{n+1} \) then \( S(cn_0 \cdots cn_n cn_{n+1}) \) for exactly one \( cn_{n+1} \).

Then a scheduled type 2 semantics is a transition system on executions \( \epsilon = cn_0 \cdots cn_n \) such that \( \epsilon \rightarrow \epsilon' \) if and only if \( \epsilon' = cn_0 \cdots cn_n cn_{n+1} \) and \( S(\epsilon') \).

Define now the barbed simulation preorder \( \sqsubseteq \) on executions by requiring the existence of a witness relation \( R \) which satisfies reduction closure and barb preservation (when \( cn_0 \cdots cn_n \downarrow \text{obs} \) if and only if \( cn_n \downarrow \text{obs} \)), but not necessarily symmetry. We immediately obtain from Theorem 3.6.4 the following corollary:

**Corollary 3.7.1.** For all well-formed type 1 configurations \( cn \) on a connected network graph,

\[
\langle \text{down}(cn) \rangle \sqsubseteq cn.
\]

**Proof.** It suffices to note that if \( \epsilon = cn_0 \cdots cn_n \rightarrow \epsilon' = cn_0 \cdots cn_{n+1} \) then \( cn_n \rightarrow cn_{n+1} \), and if \( \epsilon \downarrow \text{obs} \) then \( cn_n \downarrow \text{obs} \) as well. \(\square\)
Chapter 4

**ABS-NET: Fully Decentralized Runtime Adaptation for Distributed Objects**

4.1 Introduction

One motivation for decoupling computational processes and physical infrastructure in a distributed setting is that it becomes possible to handle resource allocation at layers lower than the application layer. Potentially, tasks can then be performed at the physical machine most suited at the moment, continually meeting global system requirements for e.g. even utilization and task-local requirements such as a limited response time.

We consider the problem of runtime adaptation of tasks in the context of the Core ABS fragment of ABS. In the semantics of Core ABS, no account is made of location or relation to a physical infrastructure. In order to account for issues related to location, object migration and physical infrastructure constraints, we extend Core ABS runtime configurations to include an abstract graph representation of a network, and extend runtime behaviour to include mobility of objects between network nodes, similarly to in previous chapters. Conceptually, adaptation is made possible by a controller process running on each network node, which communicates with controllers at other nodes but makes independent decisions on object mobility based only on local data.

To enable precise reasoning and experiments on adaptivity, we define three central Quality of Services (QoS) objectives which a solution for runtime adaptation in our context can be assessed against: node load, arc load and message latency. We abstract from many implementation-level concerns when interpreting these objectives in our setting. The load for a specific node at a specific time is simply the number of active tasks running on it. The load for a specific arc is the number of messages traversing the arc. The latency for a specific message is the number of hops needed to reach its destination. We then restrict our consideration of adaptivity to the problem of how to ship objects around to achieve the objectives as well as possible, given a specific static network topology, ABS program, and node-local procedure for managing migrations.

Using a simulator which implements the key parts of our semantics, we have investigated how well objectives are fulfilled for some application-relevant choices of network topologies, programs and migration procedures. In future work, we plan to extend the investigation to dynamic networks.

4.2 Core ABS Adaptation

The runtime unit of concurrency in Core ABS is a concurrent object group (cog). A cog contains one or more runtime objects, which perform cooperative scheduling of tasks. We use a variant of Core ABS where a single object is the unit of concurrency rather than a cog, similarly to Albert et al. [3]. The choice is motivated by our focus on network adaptability of individual objects and computation tasks, which becomes more complicated when objects in a group must perform intermittent synchronization. In this language variant, all individual objects can be interpreted as actors, having local store and communicating with the environment only via asynchronous message passing. Additionally, our language variant fixes a number of
minor inconsistencies in the syntax and semantics of the original Core ABS, for example by prohibiting multiple return statements which could cause unexpected nonterminating behaviour. The language variant is described in detail in the previous chapter, see also [25].

A fragment of a Core ABS program is given as an example in Figure 4.1. The CastNode interface defines a method aggregate, which, when called on some object, is intended to perform a convergecast operation in the binary tree rooted at that object. Specifically, this means that if an object implementing CastNode is a leaf in the tree (an instance of class LeafNode), it simply returns a locally known integer, but if the object has child nodes in the tree (an instance of class BranchNode), aggregate is called on both of those objects and the results are added to the local integer and returned. In this way, the aggregate method for the object \( o \) always returns the aggregate of all local values in the binary tree of objects rooted at \( o \).

The implementation of the aggregate method in the program highlights the use in Core ABS of futures as placeholders for results from asynchronous method calls. The variables \( f_{\text{Left}} \) and \( f_{\text{Right}} \) hold futures which ultimately resolve to integer values, as indicated by their type declarations. In the right hand side of the declarations of the futures, the delimiter ‘!’ between the object variable name and the method name signifies asynchronous invocation, which always immediately returns a future. The usual dot delimiter ‘.’ signifies a synchronous invocation which blocks the caller until the final result is returned without any intermediary.

Before returning the aggregate of the current object, the aggregate of each child node is retrieved by appending \( .\text{get} \) to the variable holding the respective future. Evaluations of assignments with this construct can be blocking, unless an \( \text{await} \) statement was executed first with the future variable involved, e.g. \( \text{await } f_{\text{Left}}?; \). Executing \( \text{await} \) when the associated future has not yet been resolved does not force the caller into busy waiting; if there are method invocations for the object waiting be processed, control can be changed to the corresponding process at the discretion of the scheduler, and pass back to the original invocation later.

Informally, a Core ABS runtime configuration is a bag of objects and futures equipped with unique identifiers, along with unprocessed method invocations. An object in the bag has values for all variables defined in its class, a queue of processes representing received method invocations, and possibly an active process. Futures either have the value to which they resolve, or a placeholder to indicate that no resolution is available. When an asynchronous method invocation statement is executed, a method invocation is added to the bag, ready to be consumed by the callee. In contrast with actor languages such as Erlang and Rebeca [80], which provide the traditional guarantee that messages from one actor to another are always processed in the order they are sent, the Core ABS semantics does not prescribe any particular order for processing method invocations. In effect, the runtime environment provides an unbounded number of one-place buffers that objects can use to communicate with objects for which identifiers are known.

While interface names are proper type names in Core ABS, class names are not, and are thus only used in object creation with the \( \text{new} \) keyword. For example, the assignment \( \text{CastNode } nd = \text{new LeafCastNode}(0); \) creates a LeafCastNode object with the \( \text{val} \) variable set to 0.

### 4.3 Network Model and Semantics

To reason about object adaptability to environmental conditions, we bring selected parts of the infrastructure of a distributed system into our model, namely, network endpoints and links. Endpoints and links are modelled as graph nodes and arcs with message queues, respectively. Conceptually, we consider a node to consist of an object layer, where local objects reside, and a node controller, which acts as a mediator between the environment and node-local objects, as illustrated in Figure 4.2. This node controller is not treated explicitly in previous chapters. The dashed arrow in the figure signifies that an object identifier is known by another object and thus can be used for method invocation. The node controller also contains logic for decision-making on adaptivity. Seen abstractly, adaptivity here becomes the problem of shipping objects around in the network graph to achieve an allocation that achieves a Quality of Service objective—with the added constraint that all reallocations must be decided locally at each node.
4.3.1 Operational Semantics

We have defined a structured operational semantics [25], building upon the functional and object levels of our variant of Core ABS, which describes the dynamics of executing an ABS model on nodes in a network at a high level of abstraction. The semantics extends the semantics of Dam [24, 26] to cover a larger fragment of Core ABS. Adaptability features such as routing information dissemination and object mobility are modelled as non-deterministic events, with the node controller consisting of nothing more than a globally unique identifier and a routing table. We refer to the combination of the Core ABS functional layer, Core ABS object syntax, the network-oriented runtime configuration syntax described below, and the associated structural operational semantics described below as ABS-NET. We intend for the semantics to both guide implementation, by defining a baseline for retaining program runtime behavioural similar to Core ABS in a distributed setting, and provide opportunities for further theoretical analysis of specific adaptability strategies by refinement.
4.3.2 Assumptions for Adaptability

In the semantics, the Core ABS program being executed is assumed to be available unaltered at all nodes. The program is therefore not explicitly represented in a runtime configuration. We consider only networks that remain static over the course of program execution. Handling benignly dynamic networks is a planned extension, but we leave all details of crash failures and byzantine failures for future work. On the same note, we also assume that messages sent between neighbour nodes cannot be lost—only ignored indefinitely as far as fairness permits.

We also assume that the behaviour of the program running on network nodes is nonterminating and cyclical. This assumption is motivated by our focus on adaptability; for adaptations to current conditions to have a chance of conveying benefits, similar conditions must hold in the future. Equivalently, if future conditions are random independently of current conditions, there is no obvious payoff in an adaptation strategy.

4.3.3 Node Controller Behaviour

The node controller’s relationship with the interpreter layer residing on the node is symbiotic. On one hand, the node controller provides message delivery services and callback functions to obtain new globally unique object identifiers for objects residing in the interpreter layer. On the other hand, the node controller triggers object movement by using callback functions that the interpreter layer makes available. We assume the node controller is aware, through its interaction with underlying network layers, of all nodes adjacent to the node it resides on, and can communicate with node controllers at neighbouring nodes. As mentioned earlier, in the model, such communication takes place through a buffer at an arc. In addition, each node controller is equipped with a self-loop arc that serves as the default route for messages that cannot immediately be routed to an adjacent node. Since there is no upper bound enforced on communication delays, the node controller always runs the risk that information received from the outside world is out of date.

The baseline behaviour for a node controller is defined by a reduction relation in the same style as Core ABS. Exchange of data with the interpreter layer, containing the local objects, is made explicit via the use of parameterized labels. For example, mobility involves an object being transported from the interpreter layer into a message passing from a node to an adjacent node.

4.3.4 Object Behaviour

Compared to Core ABS, objects at runtime are extended with input and output queues as in microABS and milliABS, as well as a map for storing resolved future values. The reduction rules in the Core ABS semantics which involve only a single object and its internal state have been transferred essentially unchanged into unlabelled interpreter layer reduction rules. In contrast, transitions that involve multiple objects or futures are translated into either transitions involving message passing or labelled rules for exchanging data with the node controller.

4.4 Adaptation

We consider three QoS objectives which runtime adaptation solutions can be assessed against: node load, arc load and message latency.

In our setting, the definition of node load is simple but coarse grained: the load on a node $u$ is the number of objects located on $u$ with active tasks. One advantage of this measure is that it is an intrinsic property of runtime configurations, rather than something extrinsic to our model such as processor load or the loadavg measure available in many Unix operating system variants. We need a model-intrinsic measure of load to enable reasoning at an abstract level about convergence to balanced allocations and that loads stay within a certain range. One disadvantage of the approach is that it fails to take into account the varying use of memory and processing power among tasks. However, in an implementation, a more fine-grained measure of load can be adopted, as long as it is linear in the number of active tasks.
We define the load of a particular arc as the number of messages traversing it. Hence, global minimization of arc load means that a minimal number of inter-node messages are sent overall, with respect to the current state of routing tables at nodes. Unless all routing tables are optimal (minimum stretch), however, there is no guarantee that the number of hops, i.e., latency, of a particular object-addressed message is minimal.

4.4.1 Node Load Balancing

Although we wish to simultaneously meet all of our QoS objectives fully, we consider node load balancing our primary concern. Load balancing solutions are also relatively well-studied in the literature, making it easier to find a good starting point.

Azar et al. [8] consider the problem of achieving balanced allocations in the framework of stochastic processes, where it is viewed as stepwise allocations of balls into bins. They highlight the use of greedy schemes for quickly converging to a ball-to-bin assignment where the maximum number of balls in any bin is minimized. The main drawback of this approach in a distributed setting is the reliance on atomic, single assignments of a ball to a bin at each algorithm step. Even-Dar and Mansour [36] study load balancing in a distributed setting where allocations are not necessarily done one-at-a-time. They give a distributed algorithm for selfish rerouting that quickly converges to a Nash equilibrium, which corresponds to a balanced resource allocation. However, at each round, locally computing a new allocation requires knowing precisely all loads in the system, which is complicated and costly to find out in the current setting.

Berenbrink et al. [11] describe and analyze fully distributed algorithms which require only local knowledge of the total number of resources and the load of one other resource to perform a single task migration step. The algorithms, some of which have attractive expected time for convergence, can be straightforwardly translated to a synchronous, round-based distributed setting, and further, e.g., via synchronizers [6], to a fully asynchronous setting. One important assumption made in the algorithm analysis is that a task can migrate to any other resource in a single concurrent round. For this property to hold, the underlying network graph must be complete, which we do not generally assume.

A factor in the convergence time is whether neutral moves are allowed, i.e., whether a migration can happen even when, as far as can be told locally, the move does not result in a more balanced allocation but merely an equally good one. If the network graph is sparse, and the number of active tasks an order magnitude greater than the number of nodes, allocations where the difference in load between any two neighbours is one but the maximal load difference is in the order of the graph diameter are possible. Such allocations clearly cannot be improved upon without neutral moves.

The problem of oscillating behaviour during task balancing can be mitigated by the use of coin flips before finalizing decisions to migrate tasks, as in the algorithms of Berenbrink et al. Oscillation can be made worse by information becoming stale, which is a fact of life in fully asynchronous systems. If the information is not too stale, however, the number of oscillation periods can sometimes be bounded [38].

4.4.2 Minimizing Communication and Other Objectives

The literature on load balancing related to scientific computing contains work on simultaneously optimizing task allocations and communication overhead. For example, Cosenza et al. [23] give a distributed load balancing scheme for simulations involving agents moving in space from worker to worker. The scheme, which is validated experimentally, optimizes both worker load and communication overhead between workers, but assumes only a small area of interest for each agent, with agents unable to communicate with other agents outside this area. In the current work, objects can communicate whenever object identifiers are known to the sender, making it harder to minimize communication overhead. Catalyurek et al. [20] describe how to use hypergraph partitioning to minimize both communication volume and migration time of tasks for parallel scientific computations. However, the repartitioning is performed in batch and requires complete, immediate knowledge of the data and computations on each node.

Querying the load of neighbours before deciding where to migrate an object can be costly in terms of arc load, and information received previously may not be accurate. Many load balancing algorithms therefore
have as a feature that the number of load queries sent is minimal when migrating a resource. A third measure which is discussed in the literature which we do not consider is the cost in terms of time and messaging for migration itself.

4.5 Evaluation

We have evaluated ABS-NET by developing a simulator for running ABS programs in a network of nodes according to our semantics. We have run the simulator with a variety of network node topologies, programs and object migration policies.

4.5.1 Simulator

Our simulator’s main purposes are to serve as a proof-of-concept for ABS-NET, and to allow us to run various adaptability case studies with particular programs and topologies. Specifically, we are interested in studying convergence properties of object migration policies in practice, and in showing that our approach of distributed execution scales to networks with many nodes. There are several other ways of executing ABS programs developed in the HATS project [22], but the main feature we need that is absent from all of them is object mobility between nodes or sites. Also, in contrast with most of these ABS backends, which aim to provide an execution platform for the full ABS language, the simulator only supports a subset of the Core ABS language; notably, the await statement is not supported.

The simulator is implemented in Java. Each node controller is implemented as a Java thread, which communicates with other controllers through TCP sockets, using the Kryonet network library [5]. One reason for choosing to use sockets is to enable to scale simulations over several physical machines and a large number of simulated network nodes. All node controllers in the network have a representation of the abstract syntax tree of the ABS program being executed, which is generated from ABS program code by the lexing and parsing frontend shared by most ABS backends.

As in the conceptual model and the formal semantics, a node controller can have zero or more objects, each having at most one active task. An active task has a reference to the statement currently being executed in the abstract syntax tree. We call an object active if it has an active task. Scheduling of active tasks is done at the node controller level in a round-robin fashion for active objects. More precisely, the scheduler deterministically steps all active tasks, checks for active objects, and then repeats the process on the new set of active tasks.

We implement statement execution by interpretation. The main reason for this choice is to enable easy serialization of objects between executing statements; to get immediate results from load balancing, we must be able to migrate active objects. One drawback of using interpretation is that local execution is slow and resource-demanding compared to the standard ABS backends.

A node controller is associated with a unique TCP port on the host system. Besides a list of neighbour handles, which abstract over underlying sockets, and a list of local objects, the node controller maintains a routing table which is broadcast to neighbours on update. Hence, except after a short interval with many updated locations, we expect routing tables to be up-to-date or nearly so. The node controller also stores incoming messages that cannot be processed locally or rerouted.

Network topology setup and program loading is handled by scripting on top of a custom simple command-line interface (CLI). When starting up, a node controller is assigned a migration policy through the CLI, which is assumed to be the same for all node controllers in the network. A migration policy is based on one of the adaptation strategies described below.

By default, the simulator starts the initial task of the initial object on a single startup node. In all our programs, the initial task creates all the objects used for the duration of the program. Migration and logging does not commence until a method with the name setupFinished is called on some object. There are several reasons for this kind of initialization; it is easier to predict load balancing behaviour with a fixed set of objects, and it is problematic to create new objects on the fly without proper distributed garbage collection, which we have not implemented.
4.5.2 Scenarios

There are many parameters to consider when setting up interesting scenarios for studying adaptation via simulations, as outlined below.

Network configuration The size and topology of the network. Large and dense networks obviously give more overhead in the form of messaging (e.g. routing and load), making simulations slower.

Object behaviour The number of objects generated by the program, intra-object communication patterns, and the fraction of objects with active tasks over time. In practice, this means selecting the appropriate ABS program and adjusting some method parameters.

Adaptation strategy This includes both the logic for deciding when and where to ship away objects, and for messaging to exchange information used as basis for decisions.

By necessity, we can explore only a small cross-section of the possible parameters, at this initial stage of the work.

Network Configurations

The possible sizes of networks to be simulated is limited by the performance of the prototype simulator. Currently, in the order of 25 network nodes can be simulated in reasonable time. On this note, we limit the evaluation to networks with three distinct underlying network topologies for nodes along the continuum from sparsely to fully connected: grids, hypergraphs and full meshes. Our base initial setup for each topology has 32 nodes.

Benchmark Programs

We have developed a number of ABS programs specifically to run in our simulator. We have avoided dynamic object creation to get as few garbage objects as possible; after initial object initialization, the number of objects remains constant. Another important constraint when developing the programs is the need, given a specific network configuration, to reason without too much difficulty about what an optimal allocation of objects to nodes looks like, based on minimizing both load and number of messages exchanged. This means that, for example, the communication graph of objects should be determinable before runtime, and most objects should be active most of the time, ensuring they are eligible for load-based migration. The programs are listed in Appendix A.

IndependentTasks.abs The starting task simply generates objects, and each generated object is called upon to perform a long-running task. There is no communication among workers—only between the coordinator object, which initializes and assigns tasks, and the generated objects. Since there is no communication, an optimal allocation is simply a completely even distribution of objects to nodes, regardless of the network topology.

Ring.abs The starting task generates objects which know the identifiers of the next object in the ring. The last object generated gets the identifier of the first object. The first object, when called, calls its next object, and so on, until the object which has the first object as next object is reached.

Star.abs The starting task generates a certain number of object star configurations, where each consists of one “center” object and one or more “fringe” objects. In each star, the latter objects communicate continually with the former object, but not with each other. There is no inter-star communication at all.
Adaptation Strategies

We have generally restricted ourselves to strategies that as a first priority balance out load evenly among nodes in the network. As a consequence, a simulator node controller continually exchanges load messages with neighbours, regardless of the specific logic for deciding when and where to move objects.

In the simulator, each migration policy defines a callback method which takes the affected node controller as a parameter. The callback method is invoked, and can possibly result in the migration of several objects to adjacent nodes.

Berenbrink et al. An adapted version of the selfish distributed load balancing algorithm by Berenbrink et al., which does not allow neutral moves. One notable difference in the simulator implementation from the abstract description given in Algorithm 1 is that only a fixed small number of objects have the possibility to migrate in each cycle, because of restrictions in the size of message buffers.

Berenbrink et al. with neutral moves An adapted version of the selfish distributed load balancing algorithm by Berenbrink et al., which does allow neutral moves, and therefore is only expected to converge to a completely stable state after a long time, exponential in the size of the network. As determined experimentally, only migrating one or two objects per cycle leads to significantly less oscillation of objects than when directly implementing the abstract description given in Algorithm 2.

Berenbrink et al. with communication intensity A variant of the preceding policy, where the objects selected for migration are selected for their affinity to the (randomly) chosen neighbour node, as determined by their communication history with objects located somewhere in the direction of the neighbour node.

Weighted neighbour load difference Once every cycle, an object and a neighbour node is chosen uniformly at random. Then, a coin biased according to the current load and the chosen neighbour’s load is flipped to decide whether the migration takes place.

Weighted neighbour load difference with communication Similar to the preceding policy, but the coin flip is also biased according to the object’s communication history in direction of chosen neighbour.

Algorithm 1 Berenbrink et al. load balancing cycle

```
for each active object o do
    u' is a neighbour chosen uniformly at random
    l is the current load
    l' is the last known load of u'
    if l > l' + 1 then
        send o to u' with probability 1 - l'/l
    end if
end for
```

Algorithm 2 Berenbrink et al. load balancing with neutral moves cycle

```
for each active object o do
    u' is a neighbour chosen uniformly at random
    l is the current load
    l' is the last known load of u'
    if l > l' then
        send o to u' with probability 1 - l'/l
    end if
end for
```
4.5.3 Scenario Objectives

Since our primary objective is to balance node load evenly, we record the load of all individual nodes over time, and then show maximum load and load standard deviation as proxy measures for how even allocations are. For scenarios with little to no object communication, this is the only measure that is relevant with respect to our objectives. For scenarios with significant messaging, we also consider the number of messages for each node between sampling intervals—with the average number of messages and standard deviation shown as proxy measures. In cases where it is clear that an optimal allocation has all communicating objects placed at one hop’s distance from each other, the number of forwarded messages can act as a proxy measure for progress towards an optimal allocation.

We sample the required quantities from simulations at a fixed global rate, corresponding roughly to a certain number of transitions (1000) in the semantics with imposed fairness via round-robin scheduling. The imposed fairness provides a degree of synchrony in the simulated network.

4.5.4 Results

Simulations of IndependentTasks.abs

The program in total creates 201 objects: one starting object which becomes inactive after initialization and 200 objects that each have a task that runs for the course of the program.

As expected, the algorithm by Berenbrink et al. without neutral moves converged very quickly and stayed unchanged with no migrations, after reaching a state where neighbour load differences have a maximum of one. For most of the runs on a 32-node hypergraph network topology, the stable state coincided with a completely balanced allocation, or very closely so. For the case of a 32-node grid, the stable allocation was frequently some distance from a fully balanced one.

The algorithm variant with neutral moves and two migrations per cycle converges to an almost-stable state quite quickly on a hypergraph, but continues to have minor oscillation of objects. With the same algorithm but five migrations allowed per cycle, there is considerably more oscillation going on after coming close to a balanced allocation. On a grid topology, where a stable allocation can be further away from a balanced allocation, allowing neutral moves gives better results than disallowing them, as expected.

The maximum load of any node and the standard deviation of node load over time for a 32-node hypergraph network topology is shown in Figure 4.3 and Figure 4.4, respectively. The corresponding measures over time for a 8x4 grid network topology are shown in Figure 4.5 and Figure 4.6, respectively. For a grid, the gain from using neutral moves is most distinctly recognized in the lower standard deviation compared to the algorithm without neutral moves in Figure 4.6.

Simulations of Star.abs

In the star program, we construct stars so that each node can hold a whole star, and there is precisely one star per network node. In an optimal allocation, therefore, there are no node-to-node message exchanges at all; all messages are sent locally.

We expected the pure load balancing policies to have markedly worse results than the policies taking inter-object communication intensity into account. In Figure 4.7, the standard deviation of the number of messages sent by nodes is shown. In Figure 4.8, the average number of sent messages over time is shown. In both cases, the measurements have been smoothed out via averaging over ten samples to avoid noise. As can be seen in the figures, there is a distinct improvement with respect to messages sent when using the algorithm by Berenbrink et al. augmented with message intensity comparisons when compared to the other policies, although it is quite far from the optimum. The algorithm using probabilistic weighting of load and messaging seems to improve the most over time, although it performs similarly to the messaging-augmented load balancing algorithm by Berenbrink et al.

With all the tested migration strategies for the hypergraph, load became evenly balanced relatively quickly, similarly to the case on a hypergraph when running IndependentTasks.abs.
Simulations of Ring.abs

When running a ring of around 130 objects on a 32-node grid, there are balanced allocations with four objects per node (possibly five for some) where all objects that communicate are on either the same node or adjacent nodes. The idea is that two of the objects on a node are part of a segment of the ring, while the other two are part of another segment coming back the other way. Such allocations lead to few inter-node messages being needed for a method invocation that involves the whole ring.

In Figure 4.9, the standard deviation of messages of a 129-object ring for a grid topology is shown. Here, both the solutions which take message intensity into account show considerable improvement over time. This is also reflected in the average number of messages sent over time shown in Figure 4.10.
4.6 Conclusions and Future Work

The evaluation suggests that it is feasible in a decentralized setting to meet the objective of balanced resource allocation, and also make headway towards the objective of minimizing communication of distributed objects. However, success for a specific migration strategy depends very much on the scenario it is used in, i.e., the underlying network topology and the specific object communication patterns.

In future work, we plan to continue the theoretical and simulation-based studies to deepen our understanding of multi-dimensional resource management, to improve the performance and accuracy of the simulator, and to investigate adaptation in dynamic networks, initially only with benign churn, i.e., with controlled startup and shutdown of nodes. One problem in doing so concerns the preservation of objects...
Figure 4.7: Std. deviation of sent messages for hypergraph in Star.abs

Figure 4.8: Avg. sent messages for hypergraph in Star.abs

Figure 4.9: Std. deviation of sent messages for grid in Ring.abs
in the face of node shutdown. With the ability to safely add and remove nodes to the network comes the possibility to ensure that all the nodes in the network have a load within some given range, while minimizing the number of nodes used.
Chapter 5

Dynamic Software Product Lines

5.1 Introduction

Various application domains require systems that run continuously and without interruption. This is particularly true in mission-critical applications and any other type of highly-available application. On the other hand, software systems need to be updated in order to keep up with changing operating environments, changing requirements, bug fixes, and expansion or removal of functionality. Such systems need the ability to update their software without interrupting the system’s execution [50].

Typical Software Product Lines (SPL) approaches do not focus on dynamic aspects, and the reconfiguration of products occurs mainly statically at development time [10]. Dynamic Software Product Lines (DSPL) enable a product to be reconfigured dynamically at runtime. Therefore, dynamic software reconfiguration is understood as the ability of reconfiguring products at runtime, that is, the transformation of a product into another valid product without any kind of interruption in the running system. The reconfiguration, in this context, would take place without the need to halt the system, recompile and redeploy. From a technical perspective, dynamic reconfiguration is a challenging task due to reasons such as ensuring that dynamically updated systems will behave correctly [47] or ensuring that no state data is lost [39]. However, even if from a technical perspective things can be changed at runtime, from the Product Line (PL) perspective only certain runtime changes may make sense and preserve the consistency of a software product. Therefore, in order to preserve the consistency of the PL products when reconfigured at runtime, there must be a way to restrict the adaptations that could be performed at runtime. Thus, the adaptations performed at runtime have to be planned in advance, and the resulting product (after the adaptations are performed) must be a consistent product following all feature model constraints and definitions. Therefore, PL reconfiguration poses even more challenges.

We use the ABS modeling framework as the foundation of our solution [30], and the existing support for product line development in terms of static product generation. Static product generation introduced support for configuring a particular SPL product at compile time by taking an ABS core model and a set of delta modules and flattening them to obtain an executable core ABS model of that single product. While static and dynamic product configuration are related concepts, they differ in one key aspect. Static product configuration always starts with the base product (represented by a core ABS model) and applies a sequence of modifications until a product specified by the product line is obtained. Dynamic product reconfiguration starts with any product already configured using the above process, and applies a set of modifications to obtain a new product (out of the set of specified products). The set of products that are configurable from a given product at runtime is constrained in the sense that they have to be explicitly defined and listed. Figure 5.1 illustrates this aspect. To support this kind of dynamic reconfiguration, ABS models need to accommodate dynamic changes in their structure and behavior. Adding this facility to ABS complements the static SPL modeling capability of ABS.

The contribution of this chapter is the added support for runtime product reconfiguration to ABS, obtained by adding a dynamic representation of product specifications and deltas, and by deferring the flattening
5.2 Related Work

The goal of this section is to describe related work in the literature regarding dynamic software update, DSPL, and also highlight the contributions that are part of the HATS framework and relate to our approach.

5.2.1 Dynamic Software Update

Dynamic software updates and their different characteristics have been widely investigated before and several challenges have been identified and solutions proposed. Hicks and Nettles [50] stated the importance of dynamic updates, particularly for applications such as financial transaction processors, telephone switches, and air traffic control systems, among others.

Multi-threaded software may pose additional challenges to dynamic updates, and Neamtiu and Hicks [67] investigated dynamic updates in this context, focusing on the problem of applying an update in a timely fashion while still producing correct behavior. Hayden et al. [48] also investigated dynamic software updates for multi-threaded software and stated, based on empirical evidence, that update points can be used without creating indefinite delays in the update process.

Another important aspect that has been investigated is the state transfer for efficient updates. State transfer techniques intend to preserve the existing state of the running system, and transfer it to the new system structure after reconfiguration. Hayden et al. [49] investigated the state update field and proposed Ekiden, a state transfer updating library for C/C++ programs. Magil et al. [65] presented a solution to the automatic checking of the state transformation by running tests on the old and new software versions separately and establishing a correlation between old and new-version objects. In a later study, Hayden et al. [47] defined the first methodology for automatically verifying the correctness of dynamic software updates.

Recent work by Wernli [85] proposed an approach using first-class contexts, called Theseus. First-class contexts make global updates unnecessary, since existing threads run to termination in an old context while new threads are started in a new, updated context.

5.2.2 Dynamic Software Product Lines

Typical SPL approaches do not focus on dynamic aspects; if a product needs to be reconfigured, the reconfiguration occurs statically at development time [10]. Dynamic SPL is an emerging field of study, and typically overlaps with other technologies, particularly self-adaptive systems [10]. Most of the work in the DSPL field focuses on managing bounded adaptivity (known and planned in advance) and not unexpected adaptations [10]. Hinchey et al. [52] highlighted the main differences between SPL and DSPL and compared four relevant points, making explicit where the difference is:

- Variability management in SPL describes different possible systems, and variability management in DSPL describes different adaptations of the system.
• The reference architecture in an SPL provides a common framework for a set of individual product architectures, and a DSPL architecture is a single system architecture, which provides a basis for all possible adaptations of the system.

• Business scoping in SPL identifies the common market for the set of products, and adaptability scoping identifies the range of adaptation the DSPL supports.

• The two-life-cycle approach in SPL describes two engineering life cycles, one for family engineering and one for application engineering. The DSPL engineering life cycle aims at the systematic development of the adaptive system, and the usage life cycle exploits adaptability in use.

Damiani and Schaefer [28] proposed a delta-oriented DSPL, with a reconfiguration automaton specifying how to switch between different feature configurations. Damiani et al. [27] also provided a formal foundation for delta-oriented DSPL.

5.2.3 Related Work within HATS

Our work has some relations with a number of other contributions inside the HATS project and those related contributions are described next.

Deliverable 3.3 - ABS Component Model

This section describes the component model extension contribution detailed in Deliverable 3.3 [31].

The goal of the component model extension is to enable dynamic reconfigurations by adding the component notion on top of the ABS objects to enhance objects and object groups with the basic elements of components (ports, bindings, consistency, and hierarchy). The enhancements proposed were: the notion of an output port distinctive from the object’s fields; the keyword Critical to annotate methods to represent that while an instance of the method is executing, it is not in a safe state to be updated; a primitive to wait for an object to be in a safe state before update; and the hierarchy of locations where components can move within the location hierarchy.

Some overlap exists between our proposal and the component model previously discussed. To avoid any duplicate effort we first analyzed the aforementioned component model before designing our contribution.

Deliverable 3.3 - Dynamic Modeling of Product Lines

An overview and formalization of dynamic product lines in the context of abstract delta modeling, was also provided in Deliverable 3.3 [31]. In this context, the DSPL takes the form of a Mealy Machine, a finite automaton with an input symbol and an output symbol on every transition. The input symbol corresponds to a feature that has been turned on or off and the output symbol corresponds to the delta that has to be applied to the current product to bring it up to date. Based on this representation of dynamic product lines, a cost model was also introduced. Monitoring a specific feature for change has a certain cost, and some features are more costly than others. Then, it was described how to optimize dynamic product lines by selectively removing transitions from them, effectively disregarding costly features until they become relevant. More details can be found in Deliverable 3.3 [31].

Deliverable 3.3 - MetaABS

To cope with the need to modify ABS code at runtime, KUL introduced a new reflective layer that allows introspection and manipulation of running code. This layer is exposed in a language extension called MetaABS (more details in Deliverable 3.3 [31]).

The purpose of MetaABS is to provide a unified interface for various runtime model analysis tasks. Adding meta-programming capabilities to ABS means that certain model analysis tasks can be encoded in ABS and carried out automatically while the model is executing. Meta-programming is generally understood as the
ability to observe and modify the structure and behavior of a program from within a program, either statically or at runtime. A meta-programming interface exposes basic elements of the programming language and the runtime environment to the programmer, enabling their inspection and modification. While it exposes these elements, it also abstracts away from their implementation.

Languages that support meta-programming commonly achieve this by providing reflection, that is, the ability of a program to inspect and modify itself at runtime. Thus, the meta-program (the program-transforming program) and the program that is transformed are the same. Reflection is decomposed into introspection, meaning the ability of a program to examine itself, and intercession, which enables a program to modify its state and behavior. In other words, introspection and intercession provide read and write access, respectively, to elements of the language. For example, the Java Reflection API is a meta-programming interface that provides methods for examining, and, to a very limited extent, modifying the runtime properties of objects including their class, interfaces, fields, and methods.

MetaABS is used in the context of our work to enable runtime reconfigurations, and the details of how MetaABS is used in this context are described in Section 5.5.1.

Deliverable 3.6 - Evolvable Systems

The summary of the work conducted in HATS WP 3 is presented in Deliverable 3.6 [33]. The goal of Deliverable 3.6 is to provide only a summary of the work conducted in other deliverables. Some topics are described, but not in depth (i.e., MetaABS is briefly presented in Deliverable 3.6, but it is presented in detail in Deliverable 3.3 [31]).

5.3 Static Variability in ABS

ABS is an object-oriented language designed to formally specify large software systems. The core of the ABS language resembles standard programming languages like Java, with concurrency and functional constructs. Besides this core language, four other elements exist in ABS to model and analyse variable systems following SPL [72] engineering practices. This work focuses on these four elements, enumerated below (together with the core module), and how they can be extended to model dynamic product reconfiguration.

**Feature model** This is used to model all products of an SPL by using features and feature attributes [21].

**Product selection** A product selection identifies an individual product that is of particular interest, defined by a valid combination of features.

**Core module** The core module consists of the classes that implement a complete product of the corresponding product line. Typically, the core module represents one product configuration, from which other products are derived (with the application of deltas).

**Delta module** This is a reusable unit of ABS code that can be applied incrementally to an ABS model to adapt its behaviour to conform to a particular product.

**Configuration** A configuration associates features to delta modules, enabling us to generate the ABS model for an individual product by naming it.

Throughout this chapter, we will explain these elements using a running example of a Chat product line, which is targeted at creating different chat clients.

5.3.1 Feature model

A feature model is described as a feature diagram, which is a tree of features with associated attributes and cardinality restrictions [77]. In our Chat product line example, the variations consist of the chat mode (textual, via voice, or via video), and the possibility of transferring files or not.
A feature diagram describes what combinations of features are valid. In our example, the diagram of the Chat product line is depicted in Figure 5.2: there must be at least a chat mode feature selected and the FileTransfer feature is optional. The two implications below the diagram capture extra constraints, namely that the Voice feature requires Text to be selected, and Video requires Voice. Our textual notation of feature diagrams and the usage of attributes are left out of this report since they do not play a role in the extension for dynamic reconfiguration of products.

5.3.2 Product selection

The syntax for describing product selections allows the enumeration of valid products from the product line. In our case, Figure 5.3 declares three products: HighEnd, Regular, and LowEnd. To produce the product Regular, for example, the features Voice and Text are selected, and the final code is generated based on the delta modules, as described in the following subsections.

```
product LowEnd (Text);
product Regular (Voice, Text);
product HighEnd (Video, Voice, Text, Files);
```

5.3.3 Core module

The core module, or core for short, defines the set of classes and interfaces that are part of a product configuration, without the need to apply any deltas. In our case, in Figure 5.4 the core module is composed of the classes that are needed in order to implement a LowEnd chat product (that only supports textual chatting as defined in the product selection in Section 5.3.2).

5.3.4 Delta module

A delta module, or delta for short, associates a name to a set of program transformations, as illustrated in Figure 5.5. These transformations are with respect to a core ABS program, although we will explore later how to describe transformations with respect to other products. The details of how the program transformations are specified are not relevant for this chapter. These program transformations express the addition, removal, or replacement of classes, interfaces, functions, methods, and fields.
interface Client {
    // other method definitions
}

interface Text extends Client {
    Unit message(Client client, String msg);
}

class ClientImpl implements Client, Text {
    Unit message(Client client, String msg) {
        // other method definitions
    }
    ...
}

Figure 5.4: Definition of core module classes and interfaces

delta DVoice {
    // modifications to the core to add voice functionality
}
delta DVideo {
    // modifications to the core to add video functionality
}

Figure 5.5: Definition of delta modules

5.3.5 Configuration

A configuration is a sequence of statements, each associating a delta to a set of features. In the example presented in Figure 5.6, the delta DVide = is associated to the Video feature, meaning that when the Video feature is selected, the DVide = delta should be applied. In fact, the set of features preceding the when keyword is a Boolean formula over features that is verified against the feature selection.

productline ChatPL;
features Text, Video, Voice, Files;
delta DVideo when Video;
delta DVoice when Voice;

Figure 5.6: Configuration of the Chat product line

5.4 Modeling DSPLs

ABS was designed from the beginning with variability support in mind and includes first-class constructs dedicated to modeling variability (such as feature models and deltas, cf. previous Section 5.3). Variability modeling was initially limited to the static level, as it was removed upon compilation. We introduce support for dynamic SPLs (DSPLs), enabling product reconfigurations at runtime. To enable DSPL models, we extend the ABS variability constructs in a conservative manner, i.e., without affecting existing SPL modeling semantics. This section describes the ABS language concepts and constructs that assist dynamic SPL engineering.

In Figure 5.7 the main aspects of the dynamic support for ABS are highlighted. Some existing aspects needed to be updated (marked in green) and new concepts were created (marked in blue). Inside the runtime circle in Figure 5.7 the arrows represent the following transitions:

From trigger to reconfiguration A trigger routine is responsible for checking periodically, based on certain monitoring variables, whether a reconfiguration should be performed or not. In this stage, the
trigger has access to all the available reconfigurations of the products (the reconfiguration logic is described in Section 5.4.1). Using our ChatPL example, in Section refsec:BuildingSelf-adaptingSystems, we will describe how this checking takes place.

**From reconfiguration to reconfigured product** MetaABS is the key actor at this point, performing the reconfiguration by applying the specific deltas defined in the DSPL configuration (described in Section 5.4.2) and applying the state transfer definitions (described in Section 5.4.3).

**From product to trigger** Each product configuration must have a trigger routine that should be periodically triggered in order to check possible reconfigurations.

These transitions will be discussed in more detail in Section 5.5.

![Figure 5.7: Dynamic aspects of ABS DSPLs](image)

### 5.4.1 Product selection

A DSPL in ABS is a set of software products that are available at runtime together with an initial product. The initial product is the product that has been configured statically and is active when the system is deployed and running. When the initial product is reconfigured dynamically, a different product becomes active and the system behaves according to the specification of the new product. That new product can be reconfigured into yet another product and so forth.

ABS requires explicitly declaring the possible dynamic transitions between products. Products of static SPLs are selected in ABS simply by associating a product name with a set of features (cf. Figure 5.3). For dynamic SPLs, the product declaration additionally lists the other products of the SPL that the given product can be transformed into at runtime.

**Syntax** A product selection includes an optional set of Adaptations. An Adaptation denotes a product into which the current product can be transformed. The by clause specifies the state update function that needs to be applied when doing this particular product transformation. How a product is to be transformed, that is, the sequence of deltas that need to be applied to the current product, is determined – as in the static context – from the product line configuration. Figure 5.8 presents the product selection grammar.
Syntax:

```
Product Selection ::= product TypeId ( FeatureSpecs ) { | Adaptation* } }
FeatureSpecs ::= FeatureSpec ( , FeatureSpec )*  
FeatureSpec ::= FID [AttributeAssignments]
AttributeAssignments ::= { AttributeAssignment ( , AttributeAssignment)* }
AttributeAssignment ::= AID = Literal
Adaptation ::= TypeId by Update ;
Update ::= TypeId
```

Figure 5.8: Product selection grammar

Example  Figure 5.9 shows a product selection example that enables runtime reconfiguration. The “low-end” chat software product implements only the Text feature. This product can be reconfigured at runtime into a “regular” chat product that additionally supports the Voice feature. The third product is a “high-end” chat system that also supports video and file transfer. Updating a system at runtime requires the transition of the system’s execution state to match the updated system structure. In ABS, state transitions are defined using state update declarations, which will be described later in Section 5.4.3.

```
product LowEnd (Text) {
  Regular by StateLowToReg;
}
product Regular (Text, Voice) {
  HighEnd by StateRegToHigh;
  LowEnd by StateRegToLow;
}
product HighEnd (Text, Voice, Video, Files) {
  Regular by StateHighToReg;
}
```

Figure 5.9: Product declarations for the dynamic chat system SPL

The possible runtime reconfiguration paths between products of the Chat product line are illustrated in Figure 5.10 and contrasted to the static configuration options.

```
    core           Regular
     /   \         /   \         /   \  
LowEnd  Regular  LowEnd  HighEnd  LowEnd  HighEnd
     \   /         \   /         \   /  
   HighEnd       HighEnd
```

Figure 5.10: Static Chat product configuration (left) and dynamic Chat product reconfiguration (right)

5.4.2 DSPL Configuration

Transforming a software product with a certain set of features into a product with a different set of features requires changing its structure and behavior. For an ABS model, this entails adding, removing, or modifying model elements such as classes, functions, and data types. As in the static setting, this is done by applying a set of deltas in sequence, except that now deltas are applied while the system is running. The sequence of deltas that needs to be applied is determined based on the product line configuration.
Compile-time product configuration always starts with a program core and applies a sequence of delta modules to that core. In ABS, this sequence is determined by the product line configuration. A configuration links a feature model, which describes the structure of an SPL, to deltas, which implement its behavior. Features and deltas are associated through application conditions, which are logical expressions over the set of features and attributes in a feature model. The collection of applicable deltas is given by the application conditions that are true for a particular feature and attribute selection.

Syntax  The grammar of delta clauses has been extended to include from application conditions. These are evaluated in the context of the current product, while the to (or when) application conditions are evaluated in the context of the target product. We only include the relevant portions of the grammar below. For the full ABS product line configuration grammar, we refer to the ABS reference manual [1]. Figure 5.11 presents the product configuration grammar.

Syntax:

\[
\begin{align*}
\delta(C) & ::= \text{delta } \Delta \text{Spec } [\text{AfterCond}] \ [\text{FromAppCond}] \ [\text{AppCond}] \\
\text{FromAppCond} & ::= \text{from } \text{BoolExp} \\
\text{AppCond} & ::= (\text{when } | \text{to } ) \text{BoolExp} \\
\text{BoolExp} & ::= \text{BoolExp } \&\& \text{BoolExp} \\
& \quad | \text{BoolExp } || \text{BoolExp} \\
& \quad | \neg \text{BoolExp} \\
& \quad | \text{( BoolExp )} \\
& \quad | \text{FID}
\end{align*}
\]

Figure 5.11: Product configuration grammar

Example  Figure 5.12 shows a product line configuration example for the Chat SPL that governs both static and dynamic (re)configuration. Lines 4–5 apply to the static configuration: the delta DVoice is applied to the core when a product includes the voice feature, and DVideo is applied when the video feature is included.

```plaintext
productline ChatPL;
features Text, Video, Voice, Files;
delta DVoice when Voice;
delta DVideo when Video;
// deltas used for runtime adaptation
delta DVoice from Text && ~Voice && ~Video when Voice;
delta DVideo from Text && Voice && ~Video when Video;
delta DNoVideo from Text && Voice && Video when Text && Voice && ~Video;
delta DNoVoice from Text && Voice && ~Video when Text && ~Voice && ~Video;
```

Figure 5.12: Configuration with deltas used at runtime

With dynamic SPLs, a system that has been configured to resemble a certain product can be re-configured, at some point after its deployment, to behave like a different product. This means that the starting point of the dynamic reconfiguration process can be any product, not just a bare-bones core product. To accommodate this fact, the scope of delta clauses in ABS has been extended to allow two application conditions: one that specifies the feature selection after reconfiguration, introduced by the when (or to) keyword and, additionally, one that is evaluated against the current feature selection (the product before reconfiguration). This additional condition is introduced with the from keyword. The delta clauses in lines 8–11 exemplify
the use of both application conditions. In line 8, the from application condition \((\text{Text} \&\& \sim \text{Voice} \&\& \sim \text{Video})\) restricts the runtime application of delta \(\text{DVoice}\) to products that have the \(\text{Text}\) feature but not \(\text{Voice}\) and \(\text{Video}\). The when application condition \((\text{Voice})\) ensures that delta \(\text{DVoice}\) is only applied if the target product will have the \(\text{Voice}\) feature.

### 5.4.3 State Transfer

In SPL engineering, configuring a system to behave like a certain product of an SPL involves assembling its code from reusable artifacts such as code modules, patches, or, as in the case of ABS, deltas. This is also true of dynamic SPLs, where behavioral units are available at runtime and can be enabled or disabled at runtime. A running system, however, also has an execution state, i.e., the collection of values assigned to variables and fields, which is commonly stored on a stack and heap. When reconfiguring a running system, these values need to be preserved or adapted to the system’s new structure according to user-defined state transformation directives. ABS manages the transformation of states upon reconfiguration using state updates.

A state update specifies how to transfer the state of objects when transitioning the system to represent a different product of the SPL. A state update has an optional pre and a post block that are executed, respectively, in the system’s pre-transformation state and in its post-transformation state. In our chat example, in Figure 5.13, two state updates are defined for two different transitions (from a regular chat product to a high-end product, and from a high-end to a regular chat product). In this example, a manipulation concerning the audio stream used is performed.

```plaintext
update StateRegToHigh;
uses Chat;

update CallImpl {
    pre: AudioStream tmp = this.singleAudioStream;
    post: this.avStreamManger.includeAudio(tmp);
}

update StateHighToReg;
uses Chat;

update CallImpl {
    pre: AudioStream tmp = this.avStreamManger.extractSingleMainAudio();
    post: this.singleAudioStream = tmp;
}
```

Figure 5.13: State update example

In Figure 5.9 it is possible to visualize which state update is run depending on the reconfiguration being performed.

### 5.5 Dynamic Reconfiguration

While the previous section introduced the language for modeling DSPLs, this section describes the mechanisms that enable dynamic reconfiguration of executing ABS models. This ranges from description of the runtime representation of the variability model via the reconfiguration process based on dynamic delta application and transfer of state to ABS’s reflective interface, which gives the user control over the reconfiguration process. This section also discusses problems and challenges that arise with runtime adaptation, such as managing processing contexts and dealing with state references that are no longer available.
5.5.1 Building Self-Adapting Systems

Systems that adapt their behavior at runtime often need to do so autonomously, by monitoring certain variables in their operating environment and adapting as these variables change. A common way to allow a system to inspect itself and its environment, and change its own behavior, is through reflection, see Section 5.2.3.

ABS includes MetaABS, a general-purpose meta-programming interface that reflects various aspects of the runtime back to the ABS program, including operations that modify these aspects. Among these aspects are the DSPLs that the system is part of and the capability to reconfigure the system into a different product of the product line.

MetaABS provides a Productline interface that gives access to operations related for querying and adapting the running software product (Figure 5.14). The Productline provides ABS methods to query the name of the current product, the products that can be obtained through reconfiguration of the current product, and an operation to actually reconfigure the current system to behave like a new product. The Productline object itself is obtained by calling the MetaABS function getPL.

We illustrate the usage of the Productline API in Figure 5.15 with the Chat product line example. The reconfiguration logic is encapsulated in a separate class that implements a Reconfigurator interface. A reconfigurator instance will run in a separate cog (i.e., concurrently to the chat functionality), monitor the network connection, and transform the running product depending on the available bandwidth.

```java
class Reconfigurator(Connection conn) {
    Unit run() {
        ProductLine pl = getPL();
        while (True) {
            String p = pl.getCurrentProduct();
            Bandwidth bw = conn.checkBandwidth();
            if (p == "RegularChat") {
                if (bw == Low) {
                    pl.configureProduct("LowEndChat");
                } else if (bw == High) {
                    pl.configureProduct("HighEndChat");
                }
            } else if (p == "HighEndChat") {
                if (bw == Low) {
                    pl.configureProduct("RegularChat");
                }
            } else if (p == "LowEndChat") {
                if (bw == Mid) {
                    pl.configureProduct("RegularChat");
                }
            } else if (p == "LowEndChat") {
                if (bw == Mid) {
                    pl.configureProduct("RegularChat");
                }
            }
        }
    }
}
```

Figure 5.15: Implementing runtime product reconfiguration for the Chat product line using MetaABS.
5.5.2 Self vs. Externally Triggered Adaptation

In the previous section, we showed how ABS models are capable of auto-reconfiguration: by using ABS’s reflective interface, the running system itself monitors certain aspects of its operating environment and autonomously reconfigures itself. Yet sometimes, changing a system’s configuration is triggered by external decisions, such as the business decision to add or remove certain features. There is no inherent obstacle to allowing ABS models to be reconfigured based on such external triggers. We have experimented with a runtime interface that receives reconfiguration directives and executes these using the same reconfiguration primitives that are accessible through the reflective MetaABS API. Another approach would be to include such an interface as a feature of the ABS system and implement it in ABS by forwarding reconfiguration commands to the reflective API.

5.5.3 Adaptive Runtime Environment

The ABS language is accompanied by a comprehensive tool framework designed for developing and analyzing various aspects of ABS models. The ABS compiler includes several back ends that generate code in different formats, some more suitable for formal analysis, others more suitable for producing portable, executable code. In conjunction with extending the ABS language to support DSPL modeling, we developed the ABS dynamic Java back end. The dynamic Java back end generates standard Java code. Using the built-in compiler of the ABS tool chain, it directly compiles an ABS model to JVM bytecode. DSPL models generated using the dynamic Java back end are capable of dynamic reconfiguration while executing on the JVM.

The key idea is to use dynamic structures in the target language to represent ABS language elements. The dynamic Java back end uses Java (singleton) objects to represent interfaces, classes, methods, objects, object fields, deltas, products, etc. Such a representation trades execution performance for fully malleable ABS models.

5.6 Conclusion

The demand to run systems continuously without interruption is increasing every day. In addition, the need to quickly respond to changes according to changing requirements/environments/conditions is also a vital need and has been gaining a lot of attention. The absence of such capabilities forces companies to follow different approaches in order to adapt their products quickly, and with the minimum interruption time possible. However, such approaches are typically very effort-intensive and error-prone.

In DSPLs, the main goal is to explore the already existing variation inside the products in order to support adaptations at runtime. DSPLs provide an interesting setting to apply the concept of dynamic updates by taking advantage of the existing variation. In this context, being able to explore existing variability at runtime requires the adaptations to be carefully planned and defined. Besides, product consistency according to the feature model (after reconfiguration) is a major concern.

Therefore, the contribution of this task was to extend the static capabilities of the ABS language and SPL support to enable dynamic aspects in the SPL development aligned with the concept of DSPLs. The Product Selection was extended in order to include also the definitions of possible reconfigurations, and the Product Configuration was extended to establish relations among the reconfigurations and the deltas. Besides, the concept of state transfer updates was defined, with the intent to preserve the states of the running products. To support the dynamic application of deltas, MetaABS was also extended and used as the foundation of the dynamic reconfiguration.
Chapter 6

Conclusion

In this deliverable we have studied autonomous evolution of software system from two perspectives:

1. From the point of view of performance adaptation, where the task is to fit a statically known software configuration to a network such that (1) functional behaviour is unaffected, and (2) various performance parameters such as processor load or communication “intensity” are optimized.

2. From the point of view of functional adaptation, where the task is to reconfigure a running system without having to recompile or redeploy the system.

Many issues remain to be resolved. We start by discussing performance adaptation and ABS-NET.

As the model stands we only study adaptation of a static software system to a static processor configuration. Approaches to handle dynamically evolving software are proposed in several HATS deliverables, including the present, and these are to a large extent orthogonal to the performance adaptation issues studied in ABS-NET. To support dynamically changing execution platforms, such as dynamically changing network configurations, more work is needed. In preliminary work not reported here we have developed a model of benignly dynamic networks, essentially to study power adaptation by node shutdown/power up. Beyond this, crash failures and byzantine failures deserve to be studied in an ABS-NET context in the future.

Within the HATS project the notion of deployment component [58, 59, 57] has been proposed in D2.1 as an approach to make resource limitations reflecting, e.g., time or processor capacity, available at the source code (ABS) level. Usually, these resource limitations are modeled in the abstract, without tying to real time, or real processor utilization, which has the advantage that these quantities can be measured and acted upon to an explicitly programmable level of precision. Reflection then becomes available to program resource management strategies directly in ABS itself. The same type of reflection could be done for ABS-NET as well, and preliminary work at UIO and KTH indicates that this is a potentially fruitful avenue for further research.

A further interesting topic is the practicality of the ABS-NET model. Even if the model itself is strongly motivated by practical, cloud-like scenarios (collections of processes executing transparently on top of a processor network) and could (and should) in principle be implemented on top of a cloud infrastructure, we are as yet only able to address a small part of the performance tuning spectrum in practice. At this point we have really only studied performance tuning by object (VM) migration. Other parameters are equally important, such as adaptive processor scheduling, buffer management (not least!), and processor and channel bandwidth constraints, to name just a few. How to automatically tune performance in such a highly dimensional parameter space in a fully decentralized fashion is a problem we have only managed to scratch the surface of, and it remains one of major research challenges in the area of systems and network management.

In the area of functional adaptation we have demonstrated the applicability of one tool (MetaABS) developed within HATS, to support controlled dynamic software adaptation within a software product line (SPL) context, and a small supporting language to define product selection strategies has been developed. This work indicates one possible approach for runtime software evolution, by using reflection to examine SPL
constraints at runtime, and to reconnect program deltas. The approach leverages existing SPLs and exploits their variability to support a similar variability within having to resort to reconfiguring or rebuilding the system, but because of this the approach is also limited to changes that are *foreseen* when the SPL was built. The challenge remain how to support *unforeseen* changes, or changes when the system under consideration is only partially known.


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.

**ABS-NET** A model for transparently executing ABS programs on a static network of processors.

**Adaptation** The process modifying a system or a piece of software to make it function better, faster, more safely, more securely, or with less resource consumption.

**Barbed equivalence** A notion of equivalence of programming language terms based on a rewrite relation with observability predicates defined on the terms.

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

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

**Compiler back end** The functional entity of a compiler that is mainly concerned with generating code for a specific machine.

**Delta** Synonymous with delta module.

**Delta module** A specification of modifications to core ABS language elements (classes, methods, interfaces, etc.)

**DHT** Distributed Hash Table. Distributed data structure commonly used in modern peer-to-peer networks for fast search over an unstructured index set.

**Dynamic software product line (DSPL)** A set of software products that can be adapted dynamically by adding and removing features.

**Feature** Generally, an increment in software functionality. On the level of feature models it is merely a label with no inherent semantic meaning.

**Feature model** An expression of the variability within product lines. Abstractly it may be seen as a system of constraints on the set of possible feature configurations.

**IDE** Integrated Development Environment.


**Network semantics** A semantics of a programming language given as an execution model on a message passing network of processing nodes.

**NID** Node identifier.
OID  Object identifier.

Routing  The process of determining a path for a message to take in order to find its way from one node in a network to another.

RPC  Remote Procedure Call

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

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

Software product  A software systems with a well-defined set of features.

Software product reconfiguration  The process of adding and removing features from a software system at runtime.

Software product line (SPL)  A set of software products that share a number of core properties, and differ on other aspects.

Software product line engineering  A development methodology for software product lines.

VM  Virtual machine
Appendix A

Adaptibility-Simulated Core ABS Programs

A.1 IndependentTasks.abs

```abs
module Load;
interface Load {
    Unit createLoad();
}
interface Util {
    Unit setupFinished();
}

class LoadImpl implements Load {
    Unit createLoad() {
        Int i = 0;
        while (i < 12000) {
            i = i + 1;
        }
    }
}
class UtilImpl implements Util {
    Unit setupFinished() { }
}
{
    Int i = 0;
    while (i < 200) {
        Load task = new LoadImpl();
        task!createLoad();
        i = i + 1;
    }
    Util util = new UtilImpl();
    util!setupFinished();
}
```

A.2 Ring.abs

```abs
module Ring;
```
interface RingNode {
    Int aggregate();
    Int aggregateCaller(RingNode caller);
    Unit setFollower(RingNode follower);
}

interface RingBuilder {
    RingNode buildRing(Int lengthAfterInitial);
    Unit setupFinished();
}

class RingNode(Int val, RingNode follower) implements RingNode {
    // assume ring is completed
    Int aggregate() {
        Fut<Int> fAggregate = follower!aggregateCaller(this);
        Int i = 0;
        while (i < 5) {
            i = i + 1;
        }
        Int aggregate = fAggregate.get;
        return val + aggregate;
    }
    // assume ring is completed
    Int aggregateCaller(RingNode caller) {
        Int aggregate = val;
        if (follower != caller) {
            Fut<Int> fFollower = follower!aggregateCaller(caller);
            Int i = 0;
            while (i < 10) {
                i = i + 1;
            }
            Int aggregateFollower = fFollower.get;
            aggregate = aggregate + aggregateFollower;
        }
        return aggregate;
    }
    Unit setFollower(RingNode node) {
        this.follower = node;
    }
}

class RingBuilder implements RingBuilder {
    // assume lengthAfterInitial >= 0
    // ensure all follower identifiers are set
    RingNode buildRing(Int lengthAfterInitial) {
        RingNode initNode = new RingNode(0, null);
        RingNode prevNode = initNode;
        Int currLength = 0;
        while (currLength < lengthAfterInitial) {
            RingNode currNode = new RingNode(currLength + 1, prevNode);
            //Fut<Unit> fFol = currNode!setFollower(prevNode);
            //Unit u = fFol.get;
        }
    }
prevNode = currNode;
currLength = currLength + 1;
}
Fut<Unit> fFolInit = initNode.setFollower(prevNode);
Unit u = fFolInit.get;
return initNode;
}

Unit setupFinished() {
}

{
RingBuilder builder = new RingBuilder();
RingNode fst = builder.buildRing(128);
builder.setupFinished();
while (True) {
    fst.aggregate();
    Int i = 0;
    while (i < 40) {
        i = i + 1;
    }
}

A.3 Star.abs

module Star;

interface StarBuilder {
    Unit buildStars(Int stars, Int fringes);
    Unit setupFinished();
}

interface Center {
    Int value();
}

interface Fringe {
    Unit query();
}

class Center(Int val) implements Center {
    Int value() {
        Int i = 0;
        while (i < 2) {
            i = i + 1;
        }
        return val;
    }
}
class Fringe(Center c, Int val) implements Fringe {
    Unit query() {

```java
while (True) {
    Fut<Int> f = c!value();
    Int i = 0;
    while (i < 10) {
        i = i + 1;
    }
    Int result = f.get;
}

class StarBuilder implements StarBuilder {
    Unit buildStars(Int stars, Int fringes) {
        Int i = 0;
        while (i < stars) {
            Center c = new Center(i);
            Int j = 0;
            while (j < fringes) {
                Fringe f = new Fringe(c, i);
                f!query();
                j = j + 1;
            }
            i = i + 1;
        }
    }
    Unit setupFinished() { }
}

{ StarBuilder builder = new StarBuilder();
  builder.buildStars(32, 6);
  builder!setupFinished();
}