Project N°:	**FP7-610582**
Project Acronym:	**ENVISAGE**
Project Title:	**Engineering Virtualized Services**
Instrument:	**Collaborative Project**
Scheme:	**Information & Communication Technologies**

Deliverable D1.1
D1.1 Modeling of Systems

Date of document: T12

Start date of the project: **1st October 2013** Duration: **36 months**

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

Final version

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

Executive Summary:
D1.1 Modeling of Systems

This document summarises deliverable D1.1 of project FP7-610582 (Envisage), a Collaborative Project supported by the 7th Framework Programme of the EC within the Information & Communication Technologies scheme. Full information on this project is available online at `http://www.envisage-project.eu`.

D1.1 is about modeling basic features of systems of distributed services, and addresses work package objective O1.1: Formal models of scalable service infrastructures and service discovery.

This deliverable describes extensions of the ABS modeling language along the following directions:

1. We add fault models, exceptions and fault recovery to ABS.
2. We introduce basic service discovery mechanisms based on object-oriented interfaces.
3. We introduce explicit modeling support for scalable service infrastructure such as the dynamic creation and management of virtual machines, the (re-)allocation of services, and the management of failures based on abstract failure models.

This deliverable further involves the development of flexible communication, synchronization and composition mechanisms which provide extension points for the integration of resource models (as developed in Task T1.2).

The deliverable focusses on consolidating and documenting the ABS language as it exists today, and mapping out the design space for the above extensions. The implementation process is guided by the needs of the case studies in Work Package 5.

List of Authors

Frank de Boer (CWI)
Nikolaos Bezirgiannis (CWI)
Vlad Serbanescu (CWI)
Einar Broch Johnsen (UIO)
Gianluigi Zavattaro (BOL)

Contents

Chapter 1

Technical Summary

The main technical contents of D1.1 consists of the three main appendices:

1. Service-Oriented Concepts and Object Orientation, based on one paper [9], included in Appendix A.1.
2. Fault models for concurrent objects, based on two papers: Fault Model Design Space for Cooperative Concurrency [10], included in Appendix A.2, and Erlang-style Error Recovery for Concurrent Objects with Cooperative Scheduling [4], included in Appendix A.3.
3. ABS Documentation, included in Appendix B.

The first two appendices describe the general design space of language features modeling service discovery mechanisms and fault generation and recovery mechanisms. The last appendix provides a detailed documentation of the the ABS language and its current available backends.

The remaining chapters of this deliverable provide a general overview of these appendices and the actual extensions to ABS language. In the next chapter of this deliverable we first provide an informal high-level introduction into the core ABS language. Chapter 3 provides a general overview of appendice A.1. Chapter 4 provides a general overview of appendices A.2 and A.3. Chapter 5 describes the actual choices made in extending the ABS language with service discovery, fault generation and recovery, and deployment components for the dynamic creation of virtual machines. Finally, chapter 6 briefly describes the ABS documentation.

Chapter 2

Introduction: The ABS Language

We present a brief overview of the core ABS language ABS [7]. This language is based on concurrent objects which communicate by means of asynchronous method calls and which internally support cooperative scheduling. The language consists of a functional and an imperative layer, described below.

Imperative layer. The *imperative layer* is used to describe the distributed control flow of the concurrent objects in terms of communication, synchronization, and internal computation. A program P consists of data type definitions, function definitions, interface definitions, class definitions, and a main block. Interface and class definitions, as well as method signatures have a standard syntax, resembling Java. Interfaces may extend other interfaces, in which case they include the signatures of their super-interfaces. Statements have access to the attributes of the current class, locally defined variables, and the formal parameters of the method they describe. Types and implementations are kept distinct: only interfaces can be used to type object references; classes are *not* types and are used to create object instances. ABS interfaces ensure strong encapsulation of implementation details so the language does not need any other mechanism for hiding.

Concurrent objects execute processes from a process queue. These processes stem from method activations. The concurrent objects we consider support *cooperative scheduling*, which means that inside the object's monitor an active process can choose to explicitly suspend its execution to allow another process from the queue to execute. This way, the interleaving of processes inside a concurrent object is textually controlled by the programmer, yet flexible and state-dependent interleaving is supported; for example, a process may suspend its execution while waiting for a reply to a method call. So-called "concurrent object groups" (gocs) define dynamic groups of objects which share control.

Asynchronous method calls generate futures [2] which are dynamically generated entities used to store the return value. Guards describe the conditions for state-dependent suspension, these are conjunctions of Boolean expressions and reply guards which check a future for the presence of a return value. Await statements suspend the current process if their guard evaluates to **false** (otherwise execution continues normally).

Functional layer. This layer provides a high-level declarative way to describe computation which abstracts from possible imperative implementations of data structures. The ground types are the basic types (Booleans, numbers, strings), interfaces, and data types which may have type parameters. Types may be type variables (i.e., uninterpreted type names [12]), ground types, or polymorphic data types. Data type definitions associate a type name to constructors. Function definitions associate a name and a return type to an expression. The expression is evaluated in the scope where the typed local variables are bound to the function's arguments. Functions may be polymorphic and require that the types are instantiated to ground types. Expressions may be expressions of the basic types, ground constructor terms, or the application of a constructor or a function to a list of expressions. The functional language further supports pattern matching with a **case**-expression which matches an expression against a list of branches and can introduce new local variables.

Chapter 3

Service-Oriented Concepts and Object Orientation

3.1 Introduction

Services are autonomous, self-describing, technology-neutral software units that can be published, discovered, queried, and composed into software applications at runtime. Designing and composing software services to form applications or composite services, require abstractions beyond those found in typical object-oriented programming languages. UIO has studied the integration of service-oriented concepts in an object-oriented kernel language [9].

This chapter gives an overview of one possible solution in the design space for service discovery semantics in object-oriented languages. We are interested in service-oriented concepts such as dynamically replacing the implementations of an announced service by another implementation, allowing the client to dynamically discover service providers and to query a service about its supported interfaces, etc.

3.2 Groups and Services

We investigate these concepts by developing a formal model of dynamic object-oriented groups which offer services to their environment. These groups fit directly into the object-oriented paradigm in the sense that they can be dynamically created, they have an identity, and they can receive method calls. In contrast to objects, groups are not used for structuring code. A group exports its services through interfaces and relies on objects to implement these services. Objects may join or leave different groups. Groups may dynamically export new interfaces, they support service discovery, and they can be queried at runtime for the interfaces they support. We define an operational semantics and a static type system for this model of dynamic object groups, and show that well-typed programs do not cause method-not-understood errors at runtime.

We study an integration of service-oriented abstractions in an object-oriented setting by defining a kernel object-oriented language with a Java-like syntax, in the style of Featherweight Java [5]. In contrast to Featherweight Java, types are different from classes in this language: interfaces describe services as sets of method signatures and classes generate objects which implement interfaces. By programming to interfaces, the client need not know how a service is implemented. For this reason, the language has a notion of *group* which dynamically connects interfaces to implementations. Groups are first-class citizens; they have identities and may be passed around. An object may dynamically join a group and thereby add new services to this group, extending the group's supported interfaces. Objects may belong to several groups. Both objects and groups may join and leave groups, thereby migrating their services between groups. Groups offer distribution of work between the implementations of the group's services, because a request to the group can be handled by any object in the group.

Syntactic Categories.	*Definitions.*	
C : Class name	P	$::= \overline{IF}\ \overline{CL}\ B$
I : Interface name	T	$::= \mathsf{Void} \mid \mathsf{Bool} \mid \mathsf{Any} \mid I \mid \mathtt{Group}(\overline{I})$
T : Type name	IF	$::= \textsf{interface}\ I\ \textsf{extends}\ \overline{I}\ \{\overline{Sg}\}$
m : Method name	CL	$::= \textsf{class}\ C(\overline{T}\ \overline{w})\ \textsf{implements}\ \overline{I}\ \{\overline{T}\,\overline{w};\ B\ \overline{M}\}$
x : Variable	Sg	$::= T\,m\,([\overline{T}\,\overline{z}])$
z : Local variable	M	$::= Sg\,B$
w : Field	B	$::= \{\overline{T}\,\overline{z}; sr;\}$
f : Field reference	e	$::= v \mid x \mid \mathtt{this}$
e : Expression	f	$::= \mathtt{this}.w$
v : Value	x	$::= f \mid z$
s : Statement list	rhs	$::= e \mid \textsf{new}\ C(\overline{e}) \mid \textsf{newgroup} \mid [\textsf{yield}]\, x.m(\overline{e})$
	$trial$	$::= z\ \textsf{subtypeOf}\ I \mid x = \textsf{acquire}\ I\ \textsf{in}\ x\ \textsf{except}\ \overline{x} \mid x\ \textsf{leaves}\ x\ \textsf{as}\ \overline{I}$
	s	$::= \textsf{skip} \mid x = rhs \mid s; s \mid \textsf{while}\ e\ \{s;\} \mid \textsf{if}\ e\ \{s;\}\ \textsf{else}\ \{s;\}$
		$\mid \textsf{try}\ trial\ \{s;\}\ \textsf{else}\ \{s;\} \mid x\ \textsf{joins}\ z\ \textsf{as}\ \overline{I} \mid \textsf{spawn}\ x.m(\overline{e})$
	sr	$::= [s;]\ \textsf{return}\ e$

Figure 3.1: Kernel language syntax. Square brackets [ ] denote optional elements, and overline denotes repetition (e.g., lists).

3.2.1 Syntax

The syntax of the studied language is given in Figure 3.1. A group is a dynamically created abstraction to which interfaces may be added over time. This way, its type may change during execution. Objects may make method calls to either other objects or to groups (in fact, the client need not know that it is calling on a group). We omit the detailed explanation of the kernel language (all details are provided in the attached paper [9]), and focus on the language primitives studied in this work:

- The *group creation* expression **newgroup** dynamically creates a new, empty group which does not offer any service to the environment.
- The optional keyword **yield** to a method call indicates a *delegation* mechanism such that the caller releases its lock for the duration of the method call.
- The trial z **subtypeOf** I is used to *query* a known group z about its supported interfaces. The query succeeds if z offers I (or a better interface), in which case the expanded knowledge of the group z becomes available.
- The trial $y =$ **acquire** I **in** x **except** $\overline{x}$ tries to find some group g or object o which is in the group x and which offers a service better than I (in the sense of subtyping) and which is not in the set $\overline{x}$.
- Objects and groups x_1 may try to withdraw services $\overline{I}$ from a group x_2 by the statement **try** x_1 **leaves** x_2 **as** $\overline{I}$ $\{s_1;\}$ **else** $\{s_2;\}$. The withdrawal only succeeds if x_2 continues to offer all the interfaces of $\overline{I}$, exported by other objects or groups.

The paper details the formalization of these abstractions in terms of a type system and an operational semantics, and shows type soundness for the kernel language.

3.2.2 Example

We illustrate the dynamic organization of objects in groups by an example of software which provides text editing support (inspired by [13]). This software provides two interfaces: SpellChecker allows the spell-checking of a piece of text and Dictionary provides functionality to update the underlying dictionary with new words, alternate spellings, etc. Apart from an underlying shared catalog of words, these two interfaces need not share state and may be implemented by different classes. Let us assume that the overall system contains several versions of Dictionary, some of which may have an integrated SpellChecker. Consider a class

```
Group(SpellChecker,Dictionary) makeEditor() {
  Group(∅) editor; SpellChecker s; Dictionary d;
  editor = newgroup;
  try d = acquire Dictionary except Nil {skip;} else {d = new Dictionary;};
  try d subtypeOf SpellChecker {
     d joins editor as Dictionary, SpellChecker;
  } else {
     d joins editor as Dictionary;
     s = new SpellChecker();
     s joins editor as SpellChecker;
  };
  return editor;
}

Void replaceDictionary(Group(SpellChecker,Dictionary) editor, Dictionary nd){
  Dictionary od;
  nd joins editor as Dictionary;
  try od = acquire Dictionary in editor except nd {skip;} else {skip;};
  try od leaves editor as Dictionary {skip;} else {skip;};
  return void;
}
```

implementing a text editor factory, which manages groups implementing these two interfaces. The factory has two methods: makeEditor dynamically assembles such software into a text editor group and replaceDictionary allows the Dictionary to be dynamically replaced in such a group. These methods may be defined as follows:

The method makeEditor acquires a top-level service d which exports the interface Dictionary (by omitting the **in**-clause of the **acquire** statement, we mean that the service discovery happens in the global system). If d also supports the SpellChecker interface, we let d join the newly created group editor as *both* Dictionary and SpellChecker. As a consequence, the group editor will now support the two interfaces Dictionary and SpellChecker. Otherwise d joins the editor group only as Dictionary, and at this point only the interface Dictionary is supported by the group editor. In this case a new SpellChecker object is created and added to the group as SpellChecker, such that the group also supports both interfaces in this execution branch.

The method replaceDictionary will replace a Dictionary service in a text editor group. First we add a new Dictionary service nd to the editor group and then fetch an old service od in the group by means of an **acquire**, where the **except**-clause is used to avoid binding to the new service nd. Finally the old service od is removed as Dictionary in the group by a **leaves** statement. The example illustrates group management by joining and leaving mechanisms as well as service discovery.

3.3 Integration into ABS

Whereas this work has studied service-oriented mechanisms in a multithreaded setting, the adaptation of these concepts into ABS [7] does not result in particular challenges. However, to maintain the conceptual focus of ABS, we do not initially propose to include the concept of groups into the language. In order to support service discovery without the additional group-construct, we propose to support service discovery by querying the deployment components which reflect the abstract locations of the architecture in an ABS model. This is currently being implemented in the ABS backends and will be tested in the case studies.

Chapter 4

Fault Model Design Space for Cooperative Concurrency

ABS is a modeling language targeting distributed systems [7]; the language combines concurrent objects and asynchronous method calls with *cooperative scheduling* of method invocations. In ABS the basic unit of computation is the *concurrent object group* (cog): a cog provides to a group of objects a shared processor. Method invocations on an object of a cog instantiate a new task that requires the cog's processor in order to execute. Cooperative scheduling allows tasks to suspend in a controlled way at explicit points in the code, so that other tasks of the object can execute. The **suspend** and **await** statements are used to explicitly release the processor: the difference between the two statements is that **await** has an associated boolean guard expressing under which condition the task should be re-activated by the scheduler. Asynchronous method invocations are used among objects belonging to different cogs; at each asynchronous method invocation a *future* is instantiated to store the return value. Futures are first class citizens in ABS and are accessed via a **get** expression; **get** is blocking because a task, executing **get** on a future of a method invocation which has not yet completed, blocks and keeps the processor until the future is written. To avoid keeping the processor, one can use an **await** `f?` to ensure that future `f` contains a value.

It is common in the literature to distinguish errors due to the software design (sometimes called *faults*) from random errors due to hardware (sometimes called *failures*). For software deployed on a single machine, such hardware failures entail a crash of the program. A characteristic of distributed systems is that failures may be *partial* [14]; i.e., the failure may cause a node to crash or a link to be broken while the rest of the system continues to operate. In our setting, a strict separation between faults and failures may seem contrived, and we will refer to unintended behavior caused by the software or hardware as faults. A fault is *masked* if the fault is not detected by the client of the service in which the fault occurs. In hierarchical fault models, faults can propagate along the path of service requests; i.e., a fault at the server level can result in a (possibly different) fault at the client level. In a synchronous communication model, a client object can only send one method call at the time whereas in an asynchronous communication model, the client may spawn several calls. Thus, it may not be clear for a client object which of the spawned calls resulted in a specific fault in the asynchronous case. However, asynchronous method calls in ABS allow results to be *shared* before they are returned: futures are first-class citizens of ABS and may be passed around. First-class futures give rise to very flexible patterns of synchronization, but they further obfuscate the path of service requests and thus of fault propagation.

We consider how faults can be introduced into ABS in a way which is faithful to its native characteristics, and discuss the appropriate introduction of faults along three dimensions: fault representation, fault behavior, and fault propagation.

4.1 How Are Faults Represented?

Exceptions are the language entities corresponding to faults in an ABS program's execution. ABS includes two kinds of entities which in principle can be used to represent faults: *objects* and *datatypes* (datatypes [6] are part of the functional layer of ABS, and abstract simple, common structures like lists and sets).

Exceptions as Objects. Representing exceptions as objects allows for a very flexible management of faults. Indeed, in this setting exceptions would have both a mutable state and a behavior. Also, one could define new kinds of exceptions using the interface hierarchy. Finally, exceptions would have identities allowing to distinguish different instances of the same fault. However, most of these features are not needed for faults: faults are generated and consumed, but they are static and with no behavior. Representing them as objects would allow a programming style which does not match the intuition and is therefore difficult to understand. Furthermore, in ABS static verification is a main concern and semantic clarity is more needed than in other languages. For this reason we think that in the setting of ABS exceptions should not be objects.

Exceptions as Datatypes. User-defined exceptions could be added by simply defining new datatypes. When the programmer wants to catch an exception, he has to specify which types of exceptions he can catch, and do a pattern matching both on the type and on its constructor to understand which particular fault happened. This produces a syntax like:

```
try { ... }
catch(List e) {
  case(e) {
   | Empty => ...
   | Cons(v,e2) => ...
} }
catch(NullPointerException e) { ... }
catch(_ e) { ... /* capture all exceptions */ }
```

where a special syntax `_` is needed to catch exceptions of any type, since there is no hierarchy for datatypes in ABS.

4.2 What is the Behavior of Faults?

Faults interrupt the normal control flow of the program. A first issue concerning faults is how they are generated. Concerning fault management, it is a common agreement that faults are manipulated with a **try**/**catch** structure, and we do not see any reason to change this approach in our design for ABS. However, after this choice has been taken, the design space is still vast and many questions still need to be investigated.

Fault Generation. In programming languages, faults can be generated either by an explicit command such as **throw** `f`, where `f` is the raised fault, or by a normal command. For instance, when evaluating the expression `x/y` a Division by Zero exception may be raised if `y` is 0. A third kind of exception may be considered in ABS; due to distribution, an asynchronous method invocation could fail due to a failure of the callee or of the communication medium. In these cases, the **get** statement (when executed on the future corresponding to the failing asynchronous call) should raise the fault, since no return value is available. The behavior of **await** depends on its intended semantics: if executing the statement **await** `f?` means that the process whose result will be stored in `f` has successfully finished, then the **await** should generate a fault, otherwise if executing **await** `f?` gives only the guarantee that a subsequent `f.get` will not block, then all faults can be raised by **get** exclusively.

Fault Effects. ABS supports modular correctness proofs based on object invariants which should hold whenever a process releases the lock of the object. An uncaught fault releases the lock by killing the running process; in this case the invariant is not guaranteed to hold, thus the object is no longer in a valid state.

For this reason the object executing the method could be killed (or restarted from an initial "correct" state if one wants to avoid to propagate null ponter exceptions).

Effect Declaration. In classic programming languages, the only effect of an uncaught fault is to kill the running process. However, we just discussed that also killing the whole object is a possible effect. One may want to have different effects for different faults. One can have a keyword **deadly** specifying that a given exception will kill the whole object if uncaught, while the behavior of just killing the process can be considered the default behavior. We can see three possibilities here. Properties may be specified:

when an exception is declared: for instance, one may write

```
deadly exception NullPointerException
```

when an exception is raised: for instance, one may write

```
throw deadly NullPointerException
```

in the signature of the method raising the exception: for instance, one may write

```
Int calc(Int x) deadly: NullPointerException {...}
```

Notice that this last approach integrates well with the declaration of which faults a method may raise, useful to statically verify that all exceptions are caught. In fact, one could write

```
Int calc(Int x) throws: DivisionByZero,
                        deadly NullPointerException {...}
```

More in general, this approach is useful also for the programmer, in particular when using methods he did not write himself.

The *current implementation* uses the second approach: the statement `abort(e)` aborts the current process only, resulting in a proof obligation for the object invariant at the point of execution. The statement **die**`(e)`, on the other hand, kills the object denoted by **this**; all other processes scheduled to be executing on the same object are aborted as well, and future calls to the same object will result in a failure. Trying to read the value of a future containing a fault `e` is equivalent to executing **die**`(e)`.

Note that an enclosing **try/catch** block can decide to handle `e` in all the above cases, in which case no process abort or object death will occur.

4.3 How Do Faults Propagate?

We have discussed in the previous section the effect of a fault on the process or object where it is raised. However, in case of fault, in particular of uncaught fault, it is reasonable to propagate the exception also to other processes/objects related to it. In particular, possible targets for propagation are processes interested in the result of the computation.

Propagation through the Return Future. In a language with synchronous method invocation the only process that can directly access the result of the computation is the caller. However, in languages with asynchronous method invocation any process receiving the future can directly access the result of the computation. The caller may be or may not be one of them, and indeed may even terminate before the result of the computation becomes ready. Thus we discuss here notification of faults to the processes synchronizing with the future. We have two possibilities: processes may synchronize with the future either with a **get** or with an **await** statement. The case of **get** is clear: those processes are interested in the result of the computation, in case of fault no correct result is available and those processes need to be notified so that they can decide how to proceed. The natural way of being notified is that the same exception is raised by **get**. A process doing an **await** is just interested in waiting for the computation to terminate, but not in

knowing its result. Thus we claim that, if the computation terminated, either with a normal value or with an exception, the **await** should not block and the exception, if any, should not be raised. The exception would be raised only if later on a **get** on the future is performed. This approach requires to put the fault notification inside the future, and has been explored in the context of ABS in [8]. Indeed, this is also the approach of Java future library (asynchronous computation with futures has been standardized in a Java library since Java SE 5 [3]). In contrast to ABS, Java's API does not distinguish between waiting for a future to become available, and retrieving the results. In fact, no primitive like **await** is available in Java. In addition, Java's futures do not faithfully propagate exceptions: the **get** method on a faulty future always raises the same exception `ExecutionException`.

4.4 Erlang-Style Error Recovery

One example of a robust (in the sense of fault-tolerant) distributed system implementation is the Erlang OTP framework, exemplified in its `gen_server` module. The behavior of an OTP server w.rt. failures is described as follows([1, pg. 362]):

> This code provides "transaction semantics" in the server–it loops with the *original value* of `State` if an exception was raised in the handler function. But if the handler function succeeded, then it loops with the value of `NewState` provided by the handler function.

This transaction semantics is enabled by Erlang's functional nature: servers handle requests in a serialized, side-effect free manner (persistent side effects are encapsulated, e.g., in a database with its own transaction semantics), and return both a result value to the client and an updated server state to the framework.

The paper [4] describes an operational semantics and implementation of Erlang-style rollback semantics for the ABS language. The semantics are enabled by the key characteristic of ABS concurrency, namely that processes that share state can never run at the same time. This means that a cog's state can safely be rolled back to the last scheduling point after the executing process crashes, preserving the local invariant. After discussing the advantages and disadvantages of this approach, it was decided to implement conventional, i.e., manual error recovery via the familiar `try/catch/finally` construct instead. The main reasons for this decision were that rollback complicates the proof theory and thus places an undue burden on other parts of the project; and that manual error recovery and correcting actions are necessary in general to preserve global invariants, for example canceling the effects of asynchronous method calls to other parts of the system.

Another result of [4] – a description of Erlang-style process supervision and restarting of subsystems – still applies to the language and will be investigated in the case studies. In brief, supervision in ABS makes use of failure propagation through first-class futures to implement supervisors that detect crashes of other parts of the model and implement appropriate recovery strategies.

Chapter 5

ABS Extensions

In this chapter we describe briefly the main extensions of the ABS related to fault generation and recovery, deployment components and service discovery.

5.1 Faults and recovery

A feature that was previously lacking and recently added to the ABS language is the capability to generate program faults and recover from them. This language extension came as a prerequisite to the support for real-world deployments of ABS software. Faults commonly appear in real-world systems, especially in distributed settings. Therefore, a robust mechanism in the form of exceptions was designed in place.

As a starting point for adding exceptions to ABS, the project undertook a survey of the design space; a summary can be found in Chapter 4. This section describes the extension that was subsequently implemented.

To be compatible with the functional core of the language, the exception type is modelled as an Algebraic Data Type (ADT). A single *open* data type is introduced with the name `Exception`. The programmer can extend this basic data type by augmenting it with user-specific exceptions (data constructors). The ABS standard library also comes bundled with certain predefined system-level exceptions. The language, however, makes no distinction between system and user exceptions, synchronous and asynchronous exceptions. Exceptions, similar to ADTs, take 0 or more arguments as exemplified:

```
exception MyException;
exception AnotherException(Int, String, Bool);
```

Furthermore, the language treats exceptions as first-class citizens; the user can construct exception-values, assign them to variables or pass them in expressions. An exception can be explicitly raised with the `throw` statement as:

```
{
  throw AnotherException(3, "mplo");
}
```

When an exception is raised the normal flow of the program will be aborted. In order to resume execution in the current process, the user has to explicitly *handle* the exception. This is achieved with a `try-catch-finally` compound statement. Statements in the `try` block will be executed and upon a raised exception the flow of execution will be transferred to the `catch` block, so as to handle (catch) the exception.

The catch block behaves similar to the `case` statement, with the only difference that the patterns can only have the type `Exception`. Every exception pattern is tried in order and if there is a match, its associated statements will be executed.

The `catch` block is followed by an optional `finally` block of statements, that will be executed regardless of an exception happening or not.

The syntax is the following:

```
try {
 stmt1;
```

```
 stmt2;
 ....
}
catch {
  exception_pattern1 => stmt_or_block;
  exception_pattern2 => ... ;
  ...
  _ => ...
}
finally {
  stmt3;
  stmt4;
}
```

If case there is no matching exception pattern, the optional "finally" block will be executed and the exception will be propagated in turn to the parent caller, and so forth, until a match is made. In the case that the propagation reaches the top-caller in the process call-stack without a successful catch, the process will be abruptly exited. Processes that were waiting on the future of the exited process will be notified with a `ProcessExitedException`.

The associated object where the exited process was operating on will remain live. That means, all other processes of the same object will not be affected. There is, however, a consideration to introduce a special exception case (named `die`) where the object and all of its processes are also exited.

5.2 Deployment Components

This section deals with extensions to ABS that model the deployment of a system onto (physical or virtual) machines. As Envisage progresses, these deployment scenarios will also serve to model resource availability, resource distribution and usage, runtime load patterns, etc.

A Deployment Component (DC) is a minimal description of the computing resources available to an ABS application. We propose to extend this notion to allow any cloud resource that can properly be quantified (for example memory, disk, network, etc). On the other hand, and in contrast to the original specification, we restrict a DC to correspond solely to a Platform Virtual Machine (VM) — indeed, the terms DC and VM are used interchangeably by our extension. We call each deployed ABS application of a separate DC/VM, an *ABS node*. A running ABS program on the cloud will effectively form a distributed network of multiple inter-communicating ABS nodes.

The most intuitive way to introduce the DC construct into the ABS language is by modeling it as an object. An interface, named `DC` allows the dynamic creation of new DC objects. By having a common interface, the different cloud backends can agree on a base service, while still being able to provide additional functionality through distinct DC-interfaced classes. What follows is the DC interface, as declared in our extension:

```
interface IDC {
    Unit shutdown();
    Triple<Rat,Rat,Rat> load();
}
```

Minimal implementation of DC is (1) shutting down the corresponding virtual machine and (2) probing for its average *system load*, i.e. a metric for how busy the system stays in a period of time. We use the Unix-style convention of returning 3 average values of 1, 5 and 15 minutes. In the case of (1), a VM shutdown implies that its cloud resources are eventually freed. Each class interfacing DC implements a different connection to a cloud-backend (IaaS platform). The intention is that the user will not have to provide such class implementations since the implementation of deployment components will come bundled with class libraries for common cloud-backend technologies (Amazon, OpenStack, Azure, etc). An example of such a class is given leaving out its concrete implementation:

```
class DC(Int cpu, Int memory) implements IDC {
```

```
  ...
}
```

The above `DC` class corresponds to a cloud backend where the VM is parameterized by the number of cpu cores and main RAM memory. We create an object of this class by passing the number of cores and memory measured in MBs as class parameters.

After calling `shutdown()`, the DC object will point to `null`. Similar to "this" identifier, a method context contains the "thisDC" (with type `DC`) that points to the DC host of the current executing object. An ABS running node can thus control itself, getting its system load or shutting down its own machine. After creating a new DC the user has to assign new running object in it. This is achieved with the infix keyword "spawns", as in the example:

```
Interf1 o1 = dc1 spawns Cls1(params..);
o1 ! method1(params..);
this.method2(o1);
```

The newly-created object will "live" in the specified DC. The `spawns` behaves similar to the `new` keyword: it creates a new COG, initializes the object, and optionally calls its run method. The `o1` identifier in the above example is a so-called *remote object reference*, that can be called for its methods and passed around in parameters as normal. Every remote object reference is a single "address" *uniquely identified across the whole network of nodes*. It should be noted that a `spawns` call will block until the VM is up and running.

5.3 Service Discovery

Service discovery, the dynamic acquisition of a computing resource suitable to fulfill a specific task or group of tasks, serves to decouple parts of a large distributed system. As such, service discovery is of interest to the Envisage case studies since certain large, distributed system architectures can be modeled naturally in this way. This section first briefly explains the basics of service discovery and lays out the design criteria for integrating service discovery into the ABS language.

At its most basic, we see a service as a computing resource suitable to fulfill one or more specific tasks. Since in ABS tasks are modeled via method calls, it makes sense to model services as ABS interfaces and implement them using ABS objects. Note that in conventional object-oriented languages, objects and interfaces might not be sufficient, but the ABS concepts of asynchronous calls, distribution via deployment components, and safe parallel execution make ABS objects powerful enough to serve as services.

We augment the feature of "Deployment Components" with the ability of discovering available services offered by a DC. As in (Appendix A.1) we adopt the notion of a service being represented by an ABS interface. In a similar fashion, the "Object Group" is translated to a DC object.

The `acquire` , `expose`, and `unexpose` methods are added to the `DC` interface. Thus, the DC interface becomes:

```
interface IDC {
    Unit shutdown();
    Triple<Rat,Rat,Rat> load();
    A acquire<A>(A);
    Unit expose<A>(A);
    Unit unexpose<A>(A);
}
```

Both of the newly-introduced methods are parametrically-polymorphic; the programmer will instantiate their types when using them, as the following example:

```
{
DC dc1 = new NebulaDC(...);
MailService mail_server;
Fut<MailService> f = dc ! acquire(mail_server);
mail_server = f.get();
mail_server ! send_mail(...);
}
```

The acquire method takes as input a "phantom object". The object is called phantom since the object's contents or reference are not actually send; the object is there only to give hints to the ABS compiler of what is the Interface we want the acquired object to comply with. The phantom object can also be introduced with a (nullary) declaration, as in the second line above: `MailService mail_server;`

The call to acquire makes a request to the DC, asking for a reference to an (possibly remote) object that complies to such an interface/service. Upon processing the request, the DC searches through its *directory facility* for object subscriptions that support such an interface. If there is no search match, the DC will raise the `ServiceNotFoundException` and recorded in the future as a fault. If the match succeeds a reference to a complying object is returned. The returned object reference from the call to acquire can then be assigned back to the phantom object or any other object. This returned object reference is typed exclusively by the mentioned Interface and the user cannot normally know which is the actual class name (class implementation) behind it, unless this can be guessed through a method implementation.

An object can be subscribe to any DC's *directory facility* through the *expose* method. For example:

```
WebService ws = this;
dc ! expose(ws);
```

Accordingly, an object can be unsubscribed for some of its services/interfaces with the `unexpose` method:

```
AdminService a = admin_object;
MyInterface m = this;
dc ! unexpose(m);
dc ! unexpose(admin_object);
```

Following the approach of phantom objects, the arguments to expose and unexpose are appropriately type-checked against the available interfaces that the object (class) implements. If the programmer omits such a phantom definition, the compiler will compute the object's principal interface (the object passed to acquire,expose,unexpose) through type-checking. If the ABS compiler cannot compute such a principal interface, it will yield a type-checking error.

This peculiar design choice (of phantom objects) was made so as to not introduce any backwards-incompatibilities (adding interfaces as first-class citizens) and further more builtin keyword-statements. A further advantage is that the implementations of acquire and (un)expose methods can vary between DCs and thus be specific to the underlying service discovery technology of the cloud provider.

Two-times (un)exposing will not yield a runtime exception and will be silently suppressed. Each DC keeps track of its own subscribed objects and automatically unsubscribes them in case they fall out of context, i.e. they have normally or erroneously exited.

Chapter 6

ABS Documentation

The ABS Documentation provides a clear well structured and easy to use description of the main features introduced by the language in order to achieve the outcome of Task 1.1. We present the language similar to a tutorial with examples associated with every new feature, starting from basic programming techniques to features that control design and implementation of cloud applications or fault tolerance and recovery.

The documentation first presents the syntax for writing simple ABS programs, introducing names and types for all variables. Second the documentation is structured based on the different programming paradigms available in ABS with syntax and examples for every construct allowing the reader to follow and test their own code straight away. Finally the backends implemented to run ABS programs are presented along with the clear sequence of steps required to run compile and run a program in a specific backend. The live version of the ABS Documentation is available at `http://docs.abs-models.org/`.

Bibliography

[1] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2 edition, 2013.

[2] Frank de Boer, Dave Clarke, and Einar Johnsen. A complete guide to the future. In *Progr. Lang. and Systems*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer-Verlag, 2007.

[3] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[4] Georg Göri, Einar Broch Johnsen, Rudolf Schlatte, and Volker Stolz. Erlang-style error recovery for concurrent objects with cooperative scheduling. In Margaria and Steffen [11], pages 6–22. To appear.

[5] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[6] Barry Jay. Algebraic data types. In *Pattern Calculus*, pages 149–160. Springer, 2009.

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

[8] Einar Broch Johnsen, Ivan Lanese, and Gianluigi Zavattaro. Fault in the future. In *COORDINATION*, volume 6721 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2011.

[9] Einar Broch Johnsen, Olaf Owe, Dave Clarke, and Joakim Bjørk. A formal model of service-oriented dynamic object groups. *Science of Computer Programming*, 2014. To appear.

[10] Ivan Lanese, Michael Lienhardt, Mario Bravetti, Einar Broch Johnsen, Rudolf Schlatte, Volker Stolz, and Gianluigi Zavattaro. Fault model design space for cooperative concurrency. In Margaria and Steffen [11], pages 23–37. To appear.

[11] Tiziana Margaria and Bernhard Steffen, editors. *6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA'14)*, volume 8803 of *Lecture Notes in Computer Science*. Springer-Verlag, 2014. To appear.

[12] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[13] Riccardo Pucella. Towards a formalization for com part I: the primitive calculus. In Mamdouh Ibrahim and Satoshi Matsuoka, editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'02)*, pages 331–342. ACM, 2002.

[14] Jim Waldo, Geoff Wyant, Ann Wollrath, and Samuel C. Kendall. A note on distributed computing. In *MOS'96)*, volume 1222 of *Lecture Notes in Computer Science*, pages 49–64. Springer, 1997.

Glossary

Appendix A

Papers

A.1 A Formal Model of Service-Oriented Dynamic Object Groups

A Formal Model of Service-Oriented Dynamic Object Groups ☆

Einar Broch Johnsen[a,*], Olaf Owe[a], Dave Clarke[b], Joakim Bjørk[a]

[a] *University of Oslo, Norway*
[b] *Uppsala University, Sweden & KU Leuven, Belgium*

Abstract

Services are autonomous, self-describing, technology-neutral software units that can be published, discovered, queried, and composed into software applications at runtime. Designing and composing software services to form applications or composite services, require abstractions beyond those found in typical object-oriented programming languages. This paper explores service-oriented abstractions such as service adaptation, discovery, and querying in an object-oriented setting. We develop a formal model of dynamic object-oriented groups which offer services to their environment. These groups fit directly into the object-oriented paradigm in the sense that they can be dynamically created, they have an identity, and they can receive method calls. In contrast to objects, groups are not used for structuring code. A group exports its services through interfaces and relies on objects to implement these services. Objects may join or leave different groups. Groups may dynamically export new interfaces, they support service discovery, and they can be queried at runtime for the interfaces they support. We define an operational semantics and a static type system for this model of dynamic object groups, and show that well-typed programs do not cause method-not-understood errors at runtime.

☆Partly funded by the EU projects FP7-610582 ENVISAGE: Engineering Virtualized Services (`http://www.envisage-project.eu`) and FP7-612985 UPSCALE: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations (`http://www.upscale-project.eu`).

*Corresponding author

Email addresses: `einarj@ifi.uio.no` (Einar Broch Johnsen), `olaf@ifi.uio.no` (Olaf Owe), `dave.clarke@it.uu.se` (Dave Clarke), `joakimbj@ifi.uio.no` (Joakim Bjørk)

Keywords: Object orientation, Object groups, Service orientation, Multithreading, Concurrency, Types, Semantics, Type safety

1. Introduction

Good software design often advocates a loose coupling between the classes and objects making up a system. Various mechanisms have been proposed to achieve this, including programming to interfaces, object groups, and service-oriented abstractions such as service discovery. By programming to interfaces, client code can be written independently of the specific classes that implement a service, using interfaces describing the services as types in the program. Object groups loosely organize a collection of objects that are capable of addressing a range of requests, reflecting the structure of real-world groups and social organizations in which membership is dynamic [1]; e.g., subscription groups, work groups, service groups, access groups, location groups, etc. Service discovery allows suitable entities (such as objects) that provide a desired service to be found dynamically, generally based on a query on some kind of interface. An advantage of designing software using these mechanisms is that the software is more readily adaptable. In particular, the structure of the groups can change and new services can be provided to replace old ones. The queries to discover objects are based on interface rather than class, so the software implementing the interface can be dynamically replaced by newer, better versions, offering improved services.

This paper explores service-oriented abstractions such as service adaptation, discovery, and querying in an object-oriented setting. Designing software services and composing services in order to form applications or composite services require abstractions beyond those found in typical object-oriented programming languages. To this end, we develop a formal model of dynamic object-oriented groups that also play the role of service providers for their environment. These groups can be dynamically created, they have identity, and they can respond to methods calls, analogously with objects in the object-oriented paradigm. In contrast to objects, groups are not used for executing code. A group exports its services through interfaces and relies on objects to implement these services. From the perspective of client code, groups may be used as if they were objects by programming to interfaces. However, groups support service-oriented abstractions not supported by objects. In particular, groups are *self-describing* in the sense that they

may dynamically export new interfaces, they support service discovery, and they can be queried at runtime for the interfaces they support. Groups are loosely assembled from objects: objects may dynamically join or leave different groups. Mechanisms for one-to-many communication and service replication for robustness are not the main focus of our model, but are to some degree supported. In this paper we develop an operational semantics and a static type system for a kernel language which captures this model of dynamic object groups, based on interfaces, interface queries, groups, and service discovery. The type system ensures that well-typed programs do not cause method-not-understood errors at runtime.

This paper extends a paper which appeared at FOCLASA 2012 [2]. In the extended version of the paper, the formalized kernel language includes a multithreaded concurrency model with reentrant method calls and a release mechanism which makes the language more expressive. These were not part of the language considered in [2]. The extended paper further expands on the use of inner groups for group management and discusses the diversity of object groups in object-oriented systems, and how different usages fit with the proposed kernel language.

The paper is organized as follows. Section 2 presents the language syntax and a small example. A type and effect system for the language is proposed in Section 3 and an operational semantics in Section 4. Section 5 defines a runtime type system and shows that the execution of well-typed programs is type-safe. Section 6 discusses different notions of groups from the perspective of the proposed kernel language. Section 7 discusses related work and Section 8 concludes the paper. The details of the type preservation proof are given in Appendix A.

2. A Kernel Language for Dynamic Object Groups

We study an integration of service-oriented abstractions in an object-oriented setting by defining a kernel object-oriented language with a Java-like syntax, in the style of Featherweight Java [3]. In contrast to Featherweight Java, types are different from classes in this language: interfaces describe services as sets of method signatures and classes generate objects which implement interfaces. By programming to interfaces, the client need not know how a service is implemented. For this reason, the language has a notion of *group* which dynamically connects interfaces to implementations. Groups are first-class citizens; they have identities and may be passed around. An object

Syntactic Categories.	*Definitions.*	
C : Class name	P	$::= \overline{IF}\ \overline{CL}\ B$
I : Interface name	T	$::= \mathsf{Void} \mid \mathsf{Bool} \mid \mathsf{Any} \mid I \mid \mathtt{Group}(\overline{I})$
T : Type name	IF	$::= \textbf{interface}\ I\ \textbf{extends}\ \overline{I}\ \{\overline{Sg}\}$
m : Method name	CL	$::= \textbf{class}\ C(\overline{T}\ \overline{w})\ \textbf{implements}\ \overline{I}\ \{\overline{T}\,\overline{w};\ B\ \overline{M}\}$
x : Variable	Sg	$::= T\,m\,([\overline{T}\,\overline{z}])$
z : Local variable	M	$::= Sg\,B$
w : Field	B	$::= \{\overline{T}\,\overline{z};\ sr;\}$
f : Field reference	e	$::= v \mid x \mid \mathtt{this}$
e : Expression	f	$::= \mathtt{this}.w$
v : Value	x	$::= f \mid z$
s : Statement list	rhs	$::= e \mid \textbf{new}\ C(\overline{e}) \mid [\textbf{yield}]\,x.m(\overline{e}) \mid \textbf{newgroup}$
	$trial$	$::= z\ \textbf{subtypeOf}\ I \mid x = \textbf{acquire}\ I\ \textbf{in}\ x\ \textbf{except}\ \overline{x}\ \mid x\ \textbf{leaves}\ x\ \textbf{as}\ \overline{I}$
	s	$::= \textbf{skip} \mid x = rhs \mid s; s \mid \textbf{while}\ e\,\{s;\} \mid \textbf{if}\ e\,\{s;\}\ \textbf{else}\ \{s;\}$
		$\mid \textbf{try}\ trial\,\{s;\}\ \textbf{else}\ \{s;\} \mid x\ \textbf{joins}\ z\ \textbf{as}\ \overline{I} \mid \textbf{spawn}\ x.m(\overline{e})$
	sr	$::= [s;]\ \textbf{return}\ e$

Figure 1: Kernel language syntax. Square brackets [ ] denote optional elements, and overline denotes repetition (e.g., lists).

may dynamically join a group and thereby add new services to this group, extending the group's supported interfaces. Objects may belong to several groups. Both objects and groups may join and leave groups, thereby migrating their services between groups. Groups offer distribution of work between the implementations of the group's services, because a request to the group can be handled by any object in the group. To study the integration of these service-oriented abstractions, we consider a concurrent kernel language. For a seamless integration with standard object-oriented languages, the kernel language supports multithread concurrency (e.g., [4, 5]), but without shared access to objects. However, this concurrency aspect is largely orthogonal to the group abstraction, which would work equally well with the actor-like concurrency of active objects (e.g., [6, 7]).

2.1. The Syntax

The syntax of the kernel language is given in Figure 1. A type T is either a basic type, an interface describing a service, or a group of interfaces. The types T are the Null type with the value null, the unit type Void with the value void, the basic type Bool of Boolean expressions, the empty interface Any, the names I of the declared interfaces, and group types $\mathtt{Group}(\overline{I})$ which state that a group supports the set $\overline{I}$ of interfaces. The use of types is further detailed in Section 3, including the subtyping relation and the type system.

A *program* P consists of a list $\overline{IF}$ of interface declarations, a list $\overline{CL}$ of class declarations, and a main block $\{\overline{T}\ \overline{z}; s;\ \textsf{return void};\}$. We assume that classes and interfaces have distinct names. The main block introduces a scope with local variables $\overline{z}$ typed by the types $\overline{T}$, and a sequence s of program statements. We conventionally denote by $\overline{z}$ a list or set of the syntactic construct z (in this case, a local program variable), and furthermore we write $\overline{T}\ \overline{z}$ for the list of typed variable declarations $T_1\ z_1; \ldots; T_n\ z_n$ where we assume that the length of the two lists $\overline{T}$ and $\overline{z}$ is the same.

Interface declarations IF associate a name I with a set of method signatures. These method signatures may be inherited from other interfaces $\overline{I}$ or may be declared directly as $\overline{Sg}$. A method *signature* Sg associates a return type T with a name m and method parameters $\overline{z}$ with declared types $\overline{T}$.

Class declarations CL have the form $\textsf{class}\ C(\overline{T}_1\ \overline{w}_1)\ \textsf{implements}\ \overline{I}\ \{\ \overline{T}_2\ \overline{w}_2; B\ \overline{M}\}$ and associates a class name C to the services declared in the interfaces $\overline{I}$. In C, these services are realized using methods to manipulate the fields $\overline{w_2}$ of types $\overline{T_2}$. The constructor block B has the form $\{\overline{T}\ \overline{z}; s;\ \textsf{return void};\}$ and initializes the fields, based on the actual values of the formal class parameters $\overline{w}_1$ of types $\overline{T}_1$. The methods M have a signature Sg and a method body $\{\overline{T}\ \overline{z};\ s;\ \textsf{return}\ e;\ \}$ which introduces a *scope* with local variables $\overline{z}$ of types $\overline{T}$ where the sequence of statements s is executed, after which the value of the expression e is returned to the client. The value void is returned for methods with return type Void, reflecting a trivial return value. Class parameters are treated like fields with both read and write access. There is read-only access to the special variable `this` which refers to the current object. We use the syntax `this`.w to refer to fields. Thus references to local variables z and fields f are syntactically distinct.

Expressions e of the kernel language consist of program variables x and Java-like expressions for constant values v, including void as well as the Boolean values true and false. *Right-hand-side expressions* (*rhs*) cover object creation $\textsf{new}\ C(\overline{e})$ where the actual constructor parameters are given by $\overline{e}$, and method calls $[\textsf{yield}]\, x.m(\overline{e})$ where the actual method parameters are given by $\overline{e}$. The optional keyword **yield** is attached to a method call to indicate a *release* mechanism such that the caller releases its lock for the duration of the method call. Method calls are synchronous and in contrast to Java all method calls are synchronized; i.e., a caller blocks until a method returns, and a callee will only accept a remote call when it is idle. In addition, we consider an expression related to service-oriented software: the *group creation* expression **newgroup** dynamically creates a new, empty group

which does not offer any service to the environment.

The *statements* s of the kernel language include standard statements such as **skip**, assignments $x = e$, sequential composition $s_1; s_2$, conditionals, and **while**-loops. *Trial* statements are statements that may fail. A trial is embedded in a `try-else` construct such that success in the trial selects the first branch and a fail selects the `else` branch. The trial z **subtypeOf** I is used to *query* a known group z about its supported interfaces. The query succeeds if z offers I (or a better interface), in which case the expanded knowledge of the group z becomes available. For typing reasons, we here require that z is a local variable (as discussed in Section 3.3).

Service discovery is localized to a named group x: The trial $y =$ **acquire** I **in** x **except** $\overline{x}$ tries to find some group g or object o which is in the group x and which offers a service better than I (in the sense of subtyping) and which is not in the set $\overline{x}$.

Service interfaces $\overline{I}$ are *dynamically exported* through a group z by the statement x **joins** z **as** $\overline{I}$, which states that an object or group x is used to implement the interfaces $\overline{I}$ in the group z. Consequently, z will support the interfaces $\overline{I}$ after x has joined the group. Objects and groups x_1 may try to withdraw service interfaces $\overline{I}$ from a group x_2 by the statement **try** x_1 **leaves** x_2 **as** $\overline{I}$ $\{s_1;\}$ **else** $\{s_2;\}$. The withdrawal succeeds if x_2 continues to offer all the interfaces of $\overline{I}$, exported by other objects or groups. Thus, removals do not affect the type of x_2. If the removal is successful then branch s_1 is taken, otherwise s_2 is taken.

Concurrency is obtained by **new** and **spawn** statements. The former creates a new object of a given class and a new thread performing the main block of the class, whereas a **spawn** statement makes a new thread performing the given method, which must be a void method. Thus a **spawn** statement executes the call asynchronously.

2.2. Example

We illustrate the dynamic organization of objects in groups by an example of software which provides text editing support (inspired by [8]). This software provides two interfaces: SpellChecker allows the spell-checking of a piece of text and Dictionary provides functionality to update the underlying dictionary with new words, alternate spellings, etc. Apart from an underlying shared catalog of words, these two interfaces need not share state and may be implemented by different classes. Let us assume that the overall system contains several versions of Dictionary, some of which may have an integrated

SpellChecker. Consider a class implementing a text editor factory, which manages groups implementing these two interfaces. The factory has two methods: makeEditor dynamically assembles such software into a text editor group and replaceDictionary allows the Dictionary to be dynamically replaced in such a group. These methods may be defined as follows:

```
Group(SpellChecker,Dictionary) makeEditor() {
  Group(∅) editor; SpellChecker s; Dictionary d;
  editor = newgroup;
  try d = acquire Dictionary except Nil {skip;} else {d = new Dictionary;};
  try d subtypeOf SpellChecker {
     d joins editor as Dictionary, SpellChecker;
  } else {
     d joins editor as Dictionary;
     s = new SpellChecker();
     s joins editor as SpellChecker;
  };
  return editor;
}

Void replaceDictionary(Group(SpellChecker,Dictionary) editor, Dictionary nd){
  Dictionary od;
  nd joins editor as Dictionary;
  try od = acquire Dictionary in editor except nd {skip;} else {skip;};
  try od leaves editor as Dictionary {skip;} else {skip;};
  return void;
}
```

The method makeEditor acquires a top-level service d which exports the interface Dictionary (by omitting the **in**-clause of the **acquire** statement, we mean that the service discovery happens in the global system). If d also supports the SpellChecker interface, we let d join the newly created group editor as *both* Dictionary and SpellChecker. As a consequence, the group editor will now support the two interfaces Dictionary and SpellChecker. Otherwise d joins the editor group only as Dictionary, and at this point only the interface Dictionary is supported by the group editor. In this case a new SpellChecker object is created and added to the group as SpellChecker, such that the group also supports both interfaces in this execution branch.

The method replaceDictionary will replace a Dictionary service in a text editor group. First we add a new Dictionary service nd to the editor group and then fetch an old service od in the group by means of an **acquire**, where the **except**-clause is used to avoid binding to the new service nd. Finally the old

$$\text{(T-Exp)}\quad \Gamma \vdash e : \Gamma(e) \qquad \text{(T-Group)}\quad \Gamma \vdash \textbf{newgroup} : \texttt{Group}(\emptyset)$$

$$\text{(T-New)}\quad \frac{\Gamma \vdash \overline{e} : \mathit{ptypes}(C)}{\Gamma \vdash \textbf{new}\ C(\overline{e}) : C} \qquad \text{(T-Sub)}\quad \frac{T \prec T' \quad \Gamma \vdash e : T}{\Gamma \vdash e : T'}$$

Figure 2: The type system for expressions and for the creation of objects and groups.

service od is removed as Dictionary in the group by a **leaves** statement. The example illustrates group management by joining and leaving mechanisms as well as service discovery.

3. A Type and Effect System

The language distinguishes behavior from implementations by using an interface as a type that describes a service. Classes are not types in source programs. A class can implement a number of different service interfaces, so its instances can export these services to clients. A program variable typed by an interface can refer to an instance of any class that implements that interface. A group typed by $\texttt{Group}(\overline{I})$ exports the services described by the set $\overline{I}$ of interfaces to clients, so a program variable of type I may refer to the group if $I \in \overline{I}$. We denote by Void the unit type and by Any the "empty" interface, which extends no interface and declares no method signatures. A service described by an interface may consist of only some of the methods defined in a class that implements the interface, so interfaces lead to a natural notion of hiding for classes. In addition to the source program types used by the programmer, class names are used to type the self-reference **this** in the context of the specific class; i.e., a class name is used as an interface type which exports *all* the methods defined in the class.

Subtyping. The subtyping relation applies to interfaces, classes and groups, with a top element Any and bottom element Null. For interfaces, the subtype relation $\prec$ is defined as the transitive closure of the extends-relation on interfaces: if I extends J' and $J' \prec J$ or $J' = J$, then $I \prec J$. We let a class be a subtype of all its implemented interfaces. We have $\mathsf{Null} \prec X \prec \mathsf{Any}$ where X is an interface, class, or group type. A group type $\texttt{Group}(S)$ is a subtype of I if there is some $J \in S$ such that $J \prec I$. We also extend the subtype relation to lists of types in the second argument, such that $J \prec \overline{I}$ if $J \prec I$ for each $I \in \overline{I}$. The reflexive closure of $\prec$ is denoted $\preceq$. We let $\texttt{Group}(S) \preceq \texttt{Group}(S')$ if for all $J \in S'$ there is some $I \in S$ such that $I \preceq J$.

Finally, $\texttt{Group}(S) \prec \texttt{Group}(S')$ expresses that $\texttt{Group}(S) \preceq \texttt{Group}(S')$ but not $\texttt{Group}(S') \preceq \texttt{Group}(S)$.

Function types $\overline{T} \to T$ are used to type methods with parameter types $\overline{T}$ and return type T. Subtyping for function types is defined as follows: $\overline{T}' \to T' \preceq \overline{T} \to T$ if $\overline{T} \preceq \overline{T}'$ and $T' \preceq T$. We let $m_\omega \in I$ denote that interface I declares a method m with function type ω; and we let $m_\omega \preceq I$ denote $\omega \preceq \omega' \wedge m_{\omega'} \in I$ (for some ω'). Similarly, $I \preceq m_\omega$ denotes $\omega' \preceq \omega \wedge m_{\omega'} \in I$ (for some ω'). The same notation applies to classes. Finally $C \prec I$ denotes that $C \preceq m_\omega$ for each $m_\omega \in I$. Thus $C \prec I$ expresses that C offers all methods of I with the same or better function types.

Typing Contexts. A typing context Γ binds variable names to types. If Γ is a typing context, x a variable, and T a type, we denote by $dom(\Gamma)$ the set of names that are bound to types in Γ (the domain of Γ) and by $\Gamma(x)$ the type to which x is bound in Γ. Define the *update* $\Gamma[x \mapsto T]$ of a typing context Γ by $\Gamma[x \mapsto T](x) = T$ and $\Gamma[x \mapsto T](y) = \Gamma(y)$ if $y \neq x$. By extension, if $\overline{x}$ and $\overline{T}$ denote the lists $x_1, \ldots, x_n$ and $T_1, \ldots, T_n$, we may write $\Gamma[\overline{x} \mapsto \overline{T}]$ for the typing context $\Gamma[x_1 \mapsto T_1] \ldots [x_n \mapsto T_n]$ and $\Gamma[\overline{x_1} \mapsto \overline{T_1}, \overline{x_2} \mapsto \overline{T_2}]$ for $\Gamma[\overline{x_1} \mapsto \overline{T_1}][\overline{x_2} \mapsto \overline{T_2}]$. For typing contexts Γ_1 and Γ_2, we define $\Gamma_1 + \Gamma_2$ such that $\Gamma_1 + \Gamma_2(x) = \Gamma_2(x)$ if $x \in dom(\Gamma_2)$ and $\Gamma_1 + \Gamma_2(x) = \Gamma_1(x)$ if $x \notin dom(\Gamma_2)$. A typing context Γ is *better* than a typing context Δ, denoted $\Gamma \preceq \Delta$, if $dom(\Gamma) = dom(\Delta)$ and $\Gamma(x) \preceq \Delta(x)$ for all x.

The Type and Effect System. Let Γ_B denote the basic typing environment, providing types for the language constants (e.g., $\Gamma_B(\mathsf{null}) = \mathsf{Null}$, $\Gamma_B(\mathsf{void}) = \mathsf{Void}$ and $\Gamma_B(\mathsf{true}) = \mathsf{Bool}$). Programs in the kernel language are analyzed in the context of Γ_B by means of a type and effect system (e.g., [9–11]). Our effects deal with changes in the typing of local group variables only, dynamically modifying their type, depending on the program point. We do not consider changes in the typing of non-local group variables as this could cause the following soundness problem: if a field $\texttt{Group}(S)$ w could get a smaller type at the point of a local call then the called method may contain an assignment $\texttt{this}.w = e$ where e is of type $\texttt{Group}(S)$ but not the smaller type. To obtain a simple type system, we do not here allow changes in the typing of fields. A possible extension is discussed at the end of the section. The inference rules for expressions are given in Figure 2 and for statements, methods, classes, and programs in Figure 3. Following standard conventions, variables appearing in a single premise of a rule are implicitly existentially quantified.

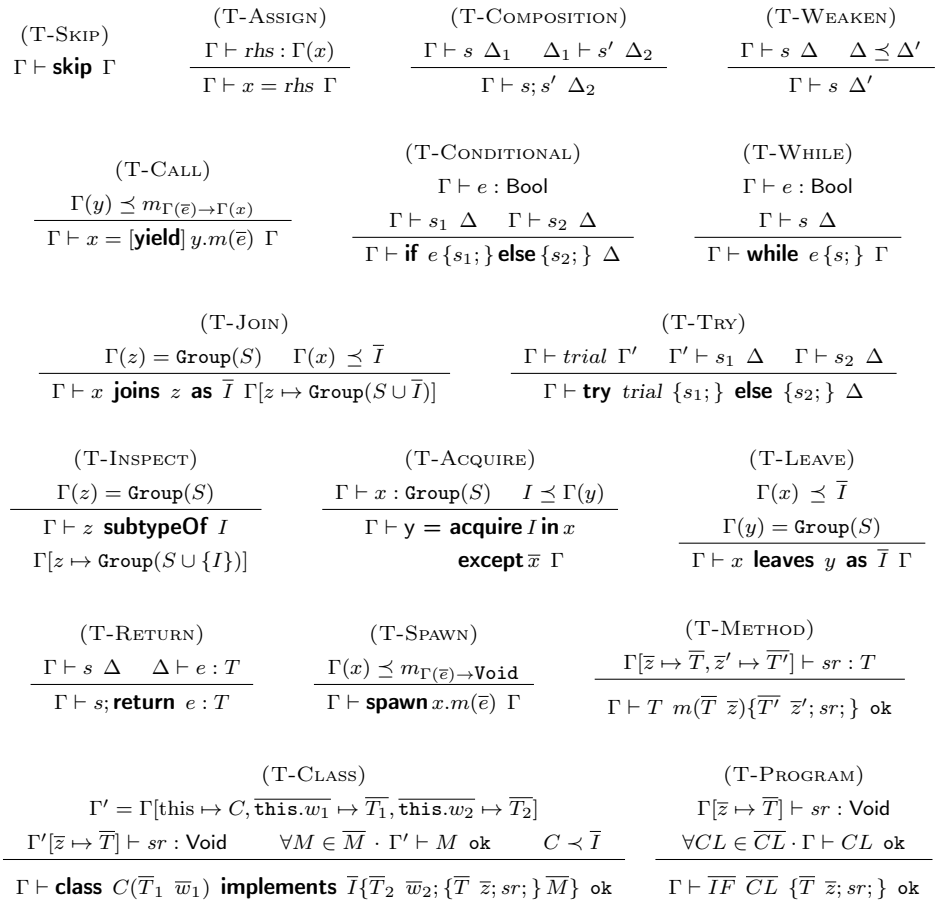

Figure 3: The type and effect system for statements, trials, methods, classes, and programs. Note that x and y denote variables and z local variables.

3.1. Expressions

Expressions are typed by the rules in Figure 2. Let Γ be a typing context. The typing judgment $\Gamma \vdash e : T$ states that the expression e has the type T if the variables in e are typed according to Γ. By T-Exp, constants and variables must be typed in Γ. By T-New, **new** C has type C if the types of the actual parameters to the class constructor can be typed to the declared types of the formal parameters of the class, as captured by the auxiliary function *ptypes*:

$$ptypes(\texttt{class}\ C(\overline{T}\ \overline{w})\ \texttt{implements}\ \overline{I}\{\overline{T}'\ \overline{w}'; B\ \overline{M}\}) = \overline{T}.$$

Note that an expression **new** C may only appear as the right-hand-side of an assignment statement; thus the type of the left-hand-side variable x will restrict C by the requirement $C \preceq \Gamma(x)$. By T-Group, a new group has the empty group type (with no exported interfaces). Rule T-Sub captures subtyping in the type system.

3.2. Statements

Statements are typed by the rules in Figure 3. Let Γ and Δ be typing contexts. The typing judgment $\Gamma \vdash s\ \Delta$ expresses that the statement or trial s is well-typed if the variables in s are typed according to Γ and that the *effect* Δ is the resulting typing context to be used for further analysis. The typing judgment $\Gamma \vdash sr : T$ expresses that the body sr is well-typed according to Γ with T as the resulting type. For a program, class, or method p, the typing judgment $\Gamma \vdash p$ `ok` means that p is well-typed according to Γ, and that the effect will not be needed in further analysis.

The typing of *standard statements* is conventional, but illustrates the effect systems. The statements **skip** and $x = e$ are typed by the rules T-Skip and T-Assign, respectively, and have no effects. The use of effects can be seen clearly in rule T-Composition, where the second statement is type checked in the typing context resulting from the first statement, and the effects are accumulated in the conclusion of the rule. Rule T-While has no effect, since no traversal of the loop is guaranteed. Rule T-Conditional propagates effects from the branches; the resulting effect is approximated by taking the intersection of the effects of the branches. Rule T-Weaken allows information to be discarded in the effect of a typing judgment; e.g., the two branches of a conditional can be unified by means of weakening.

Rule T-Try is similar to T-Conditional except that the effect of the trial is only propagated to the *then* branch, reflecting the effects of success in the trial. Rule T-Spawn is similar to T-Call except that the method must have Void as return type, since the new thread is executed asynchronously. The new thread has no effect on the current typing context.

By T-Call, a call to a method m on a variable y is well-typed if y offers an interface, say I, in which m has a function type ω such that $\omega \preceq \overline{T} \to T$, where $\overline{T}$ are the types of the actual parameters and T is the type of the left-hand-side variable. Using the keyword **yield**, a caller may release its lock for the duration of the method call. For both kinds of calls, the callee may be a group, in which case several interfaces I may satisfy the conditions above. The rule requires that there is at least one such interface I. To

ensure a type-correct binding at runtime, the static type analysis of a call $y.m(\overline{e})$ can associate the function type $\overline{T} \to T$ with the call, transforming it to $y.m_{\overline{T}\to T}(\overline{e})$. This function type gives the least type information needed to ensure well-typedness. Other forms of overloading can be considered, but these are not the focus here.

The typing of the *group manipulation statements* is as follows. By T-Acquire, service discovery is allowed on groups, with the obvious typing constraint on y. By T-Join, when an object joins a group z and contributes interfaces $\overline{I}$ to z, the type of z is extended with the interfaces $\overline{I}$ in the effect. The variable z referencing the group must be locally declared. Without this restriction, a field could dynamically extend its type, resulting in an unsound system; e.g., an assignment $f = e$ in a statically well-typed method could become unsound if the type of f were extended. However extending the type T of a local variable that copies the value of f to a type T' and assigning the result back to a field f' is allowed, as f' would need to be of the extended type T' and f would remain of type T as required by the other method. Rule T-Leave checks that the type of x is $\overline{I}$ (or better) and that y is a group, with *no overall effect*. Rule T-Inspect extends the typing context with the added information about the type of the local variable z, which must be a group.

Programs, classes, methods, and the main method of a program, are typed in the standard way. Methods do not have effects; this reflects that effects are constrained to local variables inside methods. By rule T-Return, the type of the body s; **return** e is the type of e, with no effect, but the effect of type checking s extends the typing environment for type checking of the returned expression. Likewise, classes and programs do not have effects. (For simplicity, we omit the standard type checking of interface declarations.)

Remarks. Since we associate function types with method calls, overloading is possible, and a class may even contain different implementations for the same method signature. In this case an implementation can be chosen non-deterministically, provided that the function type of the call is less than that of the chosen method.

The type system is non-deterministic due to the T-Weaken and T-Sub rules. However there is a least derivable type. The type rules may be used to define an algorithm to compute the least type. The type system in Figure 2 defines a *least type* for every expression and right-hand-side, ignoring T-Sub. Thus one may associate the least function type ω with each call. The type system in Figure 3 defines a *best effect* for every trial and statement, if T-

WEAKEN is removed and the effects of the rules for `if` and `try` statements are replaced by $\Delta_1 \cap \Delta_2$ where Δ_1 is the effect of the *then* branch and Δ_2 the effect of the *else* branch. We define $\Delta_1 \cap \Delta_2$ by $dom(\Delta_1 \cap \Delta_2) = dom(\Delta_1) \cap dom(\Delta_2)$ and by $(\Delta_1 \cap \Delta_2)(x) = \Delta_1(x) \cap \Delta_2(x)$, letting $\texttt{Group}(S_1) \cap \texttt{Group}(S_2) = \texttt{Group}(S)$ where S is the set $\{I \mid \texttt{Group}(S_1) \preceq I \wedge \texttt{Group}(S_2) \preceq I\}$ without redundant interfaces (i.e., those that extend other subtypes in the set). This is sufficient as only the typing of local group variables may change.

The least type of a local group variable is initially the declared type and it may be improved by queries and by join statements. The other basic statements maintain the type, except branching statements (`if` and `try`) which make the least type worse, but not worse than the declared type, and not worse than the type prior to the branching construct. Thus, if $z \in dom(\Gamma)$ and $\Gamma \vdash s\ \Delta$, where Δ reflect least types, then $z \in dom(\Delta)$ and $\Delta(z) \preceq \Gamma(z)$.

3.3. Discussion

The type and effect system presented above does not allow changes in the typing of fields. Changes in the typing of fields may give an unsound typing system as illustrated by the example below (where the type of a field w is strengthened in method m):

```
interface I1 extends I { ... }
interface I2 extends I { ... }
class UnSound() { I w;
 Void m1(I1 x){ this.w = x; return void; }
 Void m(I1 x){ try this.w subtypeOf I2 { this.m1(x); } else { ... }; return void; }
}
```

Here method $m1$ is type correct. In method m, the trial will improve the type of w to $I2$, but when $m1$ is executed, w will get a value of type $I1$, violating type soundness since $I1$ and $I2$ are unrelated types. As a non-local call may implicitly lead to a local call, the problem illustrated here applies to calls in general, not only local calls, i.e, a call to a method n on another object may lead to a call on method $m1$ on the current object.

However, it would be possible to allow *inspect*, *join*, and *query* statements on local as well as non-local variables. The typing of such statements may then extend the typing of fields in the same way as local variables, and such changes could be exploited inside a method. We would obtain a type sound system if we weaken the type of each field to that of its declared type after a call. This would for instance allow us to do a subtype check on a field and

then call a method of the added interface in the first branch, resuming with the declared type of all fields after the call. In the example above the type of w would be I after the call. Remark that the same call is allowed with the given type system by copying the field to a local variable before the subtype check.

Another way of achieving type soundness is to require that all local methods can be retyped in the current environment (possibly weakened as explained below). We may add a new type rule for calls and adjust the rule for methods so that it exports the non-local effects of the body:

$$\frac{\Gamma(y) \preceq m_{\Gamma(\overline{e})\to\Gamma(x)} \quad \Gamma \preceq \Gamma' \qquad \forall M \in C \cdot \Gamma' \vdash M\ \Delta \quad \Gamma \preceq \Delta}{\Gamma \vdash x = [\textsf{yield}]\mathtt{y}.m(\overline{e})\ \Delta}\ \text{(T-Call')}$$

$$\frac{\Delta \vdash e : T \qquad \Gamma[\overline{z} \mapsto \overline{T}, \overline{z}' \mapsto \overline{T'}] \vdash s\ \Delta}{\Gamma \vdash T\ m(\overline{T}\ \overline{z})\{\overline{T'}\ \overline{z}'; s; \mathtt{return}\ e;\}\ \Delta \setminus \{\overline{z}, \overline{z}'\}}\ \text{(T-Method')}$$

where C is the enclosing class, and $\Delta \setminus \{\overline{z}, \overline{z}'\}$ is Δ without the local variables $\{\overline{z}, \overline{z}'\}$. In rule T-Call' the second last premise must hold for all declared methods in *this* class (thereby ensuring type soundness in case of call-backs). With this approach, a method in C must be type checked again for each call in C, using the typing environment Γ of the respective call. It could happen that Γ is too strong (too good) for the body B to be well-typed. One may then use weakening to obtain a weaker environment Γ', in which case some of the effects prior to the call may be lost in the environment Δ after the call. A successful weakening is always possible since the method must be well-typed in the typing environment given by rule T-Class'. Thus one may weaken to that environment if no better environment applies. We assume here that Γ, Γ', and Δ have the same domains. Note that Δ cannot be stronger than the current environment Γ since there may not be any call-back. In the example of class *UnSound* one must weaken the type of w to I. (We may add a stronger rule for local calls $\mathtt{this}.m(\overline{e})$, allowing the resulting environment Δ to be stronger than Γ provided the body of m takes Γ' to Δ.)

A somewhat similar discussion applies to class initiator blocks. The effect of the initiator block on fields, without local variable bindings, is used to type the methods. Thereby each method may rely on this effect. This gives the following modification of the class rule:

Syntactic Categories.	*Definitions.*		
o : Object name	cn	::=	$\epsilon \mid o(\sigma, \delta) \mid g(export) \mid t(pr; \rho) \mid t(\texttt{idle}) \mid cn \;\; cn$
g : Group name	σ	::=	$x \mapsto (T, v) \mid \sigma + \sigma$
t : Thread name	δ	::=	$\texttt{free} \mid (t, n)$
	v	::=	$o \mid g \mid \ldots$
	$export$	::=	$\emptyset \mid \{o : I\} \mid export \cup export$
	ρ	::=	$\texttt{idle} \mid \{\sigma \mid \texttt{block}(x); sr\}; \rho$
	pr	::=	$\{\sigma \mid sr\} \mid \texttt{error}$
	s	::=	$\texttt{lock}(n) \mid \ldots$

Figure 4: The runtime syntax, extending the language syntax for values v and statements s. We let n denote a number greater than zero.

$$\frac{\Gamma' = \Gamma[\text{this} \mapsto C, \overline{\texttt{this}.w_1} \mapsto \overline{T_1}, \overline{\texttt{this}.w_2} \mapsto \overline{T_2}] \quad \Gamma'[\overline{z} \mapsto \overline{T}] \vdash s \;\; \Delta \quad \forall M \in \overline{M} \cdot (\Delta - \{\overline{z}\}) \vdash M \;\; \Delta' \quad C \prec \overline{I}}{\Gamma \vdash \textsf{class} \;\; C(\overline{T}_1 \;\; \overline{w}_1) \;\; \textsf{implements} \;\; \overline{I}\{\overline{T}_2 \;\; \overline{w}_2; \{\overline{T} \;\; \overline{z}; s; \textsf{return}\, \textsf{void}; \}\, \overline{M}\} \;\; \texttt{ok}} \;\; (\text{T-Class'})$$

4. Operational Semantics

The runtime syntax is given in Figure 4. A runtime configuration is seen in the context of the classes and interfaces defined by the given program. These definitions are fixed in our setting and give rise to auxiliary look-up functions depending on the class table, see Figure 5. A runtime configuration cn is either the empty configuration ε or a multiset of objects, groups, and threads. Objects $o(\sigma, \delta)$ have an identity o, a state σ defining the fields and class parameters, and a lock δ. We assume that the class name of an object is embedded in the object identity, such that $classOf(o)$ denotes the class of object o and $classOf(\textsf{null})$ is undefined. A state σ maps program variables x to their types T and runtime values v. The update notation of typing environments is reused for states; thus $\sigma[x \mapsto (T, v)]$ updates the binding of x in σ. At runtime, values v include object and group names, in addition to the language constants. The lock δ of an object is either `free` or a thread t has taken the lock n times, denoted (t, n). We let the predicate $\delta(t)$ denote that t has the form (t, n), expressing that the thread t holds the lock δ. Groups $g(export)$ have an identity g and contain a set $export$ of interfaces

I associated with the objects o implementing them, denoted $o : I$. Threads $t(\rho)$ have an identity t and a stack ρ of processes pr. When a thread has processes to execute, it executes the process at the top of its stack. The stack grows with method calls and shrinks at method returns. The empty stack is denoted `idle`.

A process pr has a local state σ, defining the local variables and method parameters, and a sequence sr of statements to be executed in that state, or it is the `error` process which denotes that the computation has gone wrong. To easily distinguish local states from object states, we let l denote the local state of a process, and a the state of an object. Thus $(a + l)$ represents the total state obtained from an object state a and a local state l. The look-up function for program variables x in a state σ is defined by $\sigma(x) = (T, v)$, with the corresponding projections $\sigma^{\mathcal{T}}(x) = T$ and $\sigma^{\mathcal{V}}(x) = v$ to types and values, respectively. Thus, for a state σ, $\sigma^{\mathcal{T}}$ gives the associated mapping of program variables to their current types and $\sigma^{\mathcal{V}}$ the mapping of program variables to their current values. The look-up function is extended to *values* v, such that $\sigma^{\mathcal{V}}(v) = v$ and $\sigma^{\mathcal{T}}(v)$ is the corresponding type. We reserve the name `block` for bookkeeping in the runtime typing rules (cf. Section 5), and assume that it is not in use as a variable name.

The runtime syntax extends the syntax of the surface language as follows. The runtime statement `block`(x) encodes that the process is waiting for the return value of another process (which is above it on the stack); this return value will be assigned to variable x. Observe that the syntax enforces every frame below the executing frame on a non-idle stack to start with a block statement, these are the only possible occurrences of block statements. At runtime, the statements `lock`(n) and `return` e are used to manipulate locks. As a simplifying assumption in this paper, a thread t will always lock the object in which it will execute a method activation, and unlock the object when the method activation returns. This is captured in the semantics by the premises $l^{\mathcal{V}}(\texttt{this}) = o$ and $\delta(t)$ when l is the local state of thread t and δ is the lock of object o. This assumption corresponds to synchronized methods in Java.

Structured operational semantics. The operational semantics is given by rules in the style of SOS [12], reflecting small-step semantics. Each rule describes one step in the execution of a thread. Concurrent execution is given by

$$
\begin{array}{lll}
\delta(t) & = \begin{cases} \mathit{true} & \text{if } \delta = (t, n) \\ \mathit{false} & \text{otherwise} \end{cases} \\
\mathit{init}(C, o) & = \{[\overline{z} \mapsto (\overline{T}, \texttt{default}(\overline{T})), \texttt{this} \mapsto (C, o)] \mid sr; \} \\
\mathit{atts}(C, \overline{v}) & = [\overline{\texttt{this}.w_1} \mapsto (\overline{T_1}, \overline{v}), \overline{\texttt{this}.w_2} \mapsto (\overline{T}_2, \texttt{default}(\overline{T}_2))] \\
\mathit{CT}(C) & = [\overline{\texttt{this}.w_1} \mapsto \overline{T_1}, \overline{\texttt{this}.w_2} \mapsto \overline{T_2}] \\
\mathit{export} \preceq I & = \exists\ v : J \in \mathit{export} \cdot J \preceq I \\
\mathit{export} \preceq \overline{I} & = \forall\ I \in \overline{I} \cdot \mathit{export} \preceq I \\
\mathit{export} - \overline{v} & = \{v : J \mid v : J \in \mathit{export} \wedge v \notin \overline{v}\}
\end{array}
$$

Figure 5: Auxiliary definitions in the operational semantics. Here C is $\texttt{class}\ C(\overline{T_1}\ \overline{w_1})\ \texttt{implements}\ \overline{I}\{\overline{T_2}\ \overline{w_2}; \{\overline{T}\ \overline{z}; sr; \}\ \overline{M}\}$. In addition, we use the predicate $\mathit{fresh}(t)$ to denote that thread (or group) name t is globally fresh, and $\mathit{fresh}_C(o)$ to denote that o is a globally fresh object name such that $\mathit{classOf}(\mathit{fresh}_C(o)) = C$, and $\texttt{default}(T)$ to denote some value of type T. A fresh name is not `null`.

standard SOS context and concurrency rules

$$
\frac{cn_1 \to cn_1'}{cn_1\ cn_2 \to cn_1'\ cn_2}\ (\textsc{Interleave}) \qquad \frac{cn_1 \to cn_1' \quad cn_2 \to cn_2'}{cn_1\ cn_2 \to cn_1'\ cn_2'}\ (\textsc{Parallel})
$$

We assume associative and commutative matching over configurations (as in rewriting logic [13]). Thus threads can execute in parallel in distinct parts of the configuration, which leads to the following restrictions on concurrent execution: Two threads which require the *same object* cannot execute in parallel. In a thread-based setting, a thread must take the lock of an object to access its state, and explicitly release this lock when it no longer requires the object's state. Two partial functions inc_t and dec_t capture the locking and unlocking discipline for a thread t; $\mathit{inc}_t(n, \delta)$ is applied whenever t wants to grab a lock δ (and grabs the lock n times) and $\mathit{dec}_t(\delta)$ is applied whenever t wants to release a lock δ. These functions are defined as follows:

$$
\begin{array}{ll}
\mathit{inc}_t(n, \texttt{free}) = (t, n) & \mathit{dec}_t((t, 1)) = \texttt{free} \\
\mathit{inc}_t(n, (t, n')) = (t, n + n') & \mathit{dec}_t((t, n + 1)) = (t, n)
\end{array}
$$

In the rules, if the function is undefined, then the rule cannot be applied.

The transition rules are given in Figures 6 and 7. All rules which make use of the object state require that the thread t has already taken the object lock δ, expressed by the auxiliary predicate $\delta(t)$. Rules involving a group will lock the group in question for one reduction step, thereby disallowing concurrent execution of other threads which require access to the interface

$$\text{(Skip)}\quad t(\{l \mid \texttt{skip}; sr\}; \rho) \to t(\{l \mid sr\}; \rho)$$

$$\text{(While)}\quad t(\{l \mid \texttt{while}\, e\, \{s;\}; sr\}; \rho) \to t(\{l \mid \texttt{if}\, e\, \{s; \texttt{while}\, e\, \{s;\}\} \texttt{else}\, \{\texttt{skip}\}; sr\}; \rho)$$

$$\text{(Cond1)}\quad \frac{l^{\mathcal{V}}(\texttt{this}) = o \quad \delta(t) \quad (a+l)^{\mathcal{V}}(e) = \mathit{true}}{t(\{l \mid \texttt{if}\, e\, \{s_1;\}\, \texttt{else}\, \{s_2;\}; sr\}; \rho)\ o(a,\delta) \to t(\{l \mid s_1; sr\}; \rho)\ o(a,\delta)}$$

$$\text{(Cond2)}\quad \frac{l^{\mathcal{V}}(\texttt{this}) = o \quad \delta(t) \quad (a+l)^{\mathcal{V}}(e) = \mathit{false}}{t(\{l \mid \texttt{if}\, e\, \{s_1;\}\, \texttt{else}\, \{s_2;\}; sr\}; \rho)\ o(a,\delta) \to t(\{l \mid s_2; sr\}; \rho)\ o(a,\delta)}$$

$$\text{(Assign1)}\quad \frac{l^{\mathcal{V}}(\texttt{this}) = o \quad \delta(t) \quad l^{\mathcal{T}}(z) = T \quad (a+l)^{\mathcal{V}}(e) = v}{t(\{l \mid z = e; sr\}; \rho)\ o(a,\delta) \to t(\{l[z \mapsto (T,v)] \mid sr\}; \rho)\ o(a,\delta)}$$

$$\text{(Assign2)}\quad \frac{l^{\mathcal{V}}(\texttt{this}) = o \quad \delta(t) \quad a^{\mathcal{T}}(f) = T \quad (a+l)^{\mathcal{V}}(e) = v}{t(\{l \mid f = e; sr\}; \rho)\ o(a,\delta) \to t(\{l \mid sr\}; \rho)\ o(a[f \mapsto (T,v)],\delta)}$$

$$\text{(Lock-Object)}\quad \frac{l^{\mathcal{V}}(\texttt{this}) = o \quad \delta(t) \quad \delta' = \mathit{inc}_t(n,\delta)}{t(\{l \mid \texttt{lock}(n); sr\}; \rho)\ o(a,\delta) \to t(\{l \mid sr\}; \rho)\ o(a,\delta')}$$

$$\text{(New-Group)}\quad \frac{\mathit{fresh}(g) \quad \delta(t)}{t(\{l \mid x = \texttt{newgroup}; sr\}; \rho) \to t(\{l \mid x = g; sr\}; \rho)\ g(\emptyset)}$$

$$\text{(New-Thread)}\quad \frac{\delta(t) \quad (a+l)^{\mathcal{V}}(x) = o' \quad \mathit{classOf}(o') = C \quad l^{\mathcal{V}}(\texttt{this}) = o \quad \mathit{fresh}(t') \quad pr = \mathit{bind}(m_\omega, o', C, (a+l)^{\mathcal{V}}(\overline{e}))}{t(\{l \mid \texttt{spawn}\, x.m_\omega(\overline{e}); sr\}; \rho)\ o(a,\delta) \to t(\{l \mid sr\}; \rho)\ t'(pr; \texttt{idle})\ o(a,\delta)}$$

$$\text{(New-Object)}\quad \frac{\mathit{fresh}_C(o') \quad \mathit{fresh}(t') \quad l^{\mathcal{V}}(\texttt{this}) = o \quad pr = \mathit{init}(C,o') \quad \delta(t) \quad a' = \mathit{atts}(C, (a+l)^{\mathcal{V}}(\overline{e}))}{t(\{l \mid x = \texttt{new}\ C(\overline{e}); sr\}; \rho)\ o(a,\delta) \to t(\{l \mid x = o'; sr\}; \rho)\ o(a,\delta)\ t'(pr; \texttt{idle})\ o'(a', (t',1))}$$

$$\text{(Call1)}\quad \frac{\delta(t) \quad (a+l)^{\mathcal{V}}(y) = o' \quad \mathit{classOf}(o') = C \quad l^{\mathcal{V}}(\texttt{this}) = o \quad pr = \mathit{bind}(m_\omega, o', C, (a+l)^{\mathcal{V}}(\overline{e}))}{t(\{l \mid x = y.m_\omega(\overline{e}); sr\}; \rho)\ o(a,\delta) \to t(pr; \{l \mid \texttt{block}(x); sr\}; \rho)\ o(a,\delta)}$$

$$\text{(Call2)}\quad \frac{\delta = (t,n) \quad (a+l)^{\mathcal{V}}(y) = o' \quad \mathit{classOf}(o') = C \quad l^{\mathcal{V}}(\texttt{this}) = o \quad pr = \mathit{bind}(m_\omega, o', C, (a+l)^{\mathcal{V}}(\overline{e}))}{t(\{l \mid x = \texttt{yield}\ y.m_\omega(\overline{e}); sr\}; \rho)\ o(a,\delta) \to t(pr; \{l \mid \texttt{block}(x); \texttt{lock}(n); sr\}; \rho)\ o(a, \texttt{free})}$$

Figure 6: The operational semantics (1).

(Call3)

$$\frac{l^{\mathcal{V}}(\texttt{this}) = o \quad (a+l)(y) = (\texttt{Group}(S), g) \quad I \in S \quad \delta(t) \quad v : I \in \mathit{export} \quad I \preceq m_\omega}{t(\{l \mid [\texttt{yield}]\ x = y.m_\omega(\overline{e}); sr\}; \rho)\ o(a,\delta)\ g(\mathit{export}) \to t(\{l \mid [\texttt{yield}]\ x = v.m_\omega(\overline{e}); sr\}; \rho)\ o(a,\delta)\ g(\mathit{export})}$$

(Join)

$$\frac{(a+l)^{\mathcal{V}}(x) = v \quad l(z) = (\texttt{Group}(S), g) \quad l^{\mathcal{V}}(\texttt{this}) = o \quad T = \texttt{Group}(S \cup \overline{I}) \quad \delta(t) \quad \mathit{export}' = \bigcup_{I \in \overline{I}} \{v : I\} \cup \mathit{export}}{t(\{l \mid x\ \texttt{joins}\ z\ \texttt{as}\ \overline{I}; sr\}; \rho)\ o(a,\delta)\ g(\mathit{export}) \to t(\{l[z \mapsto (T, g)] \mid sr\}; \rho)\ o(a,\delta)\ g(\mathit{export}')}$$

(Return)

$$\frac{l^{\mathcal{V}}(\texttt{this}) = o \quad \delta(t) \quad \delta' = \mathit{dec}_t(\delta) \quad (a+l)^{\mathcal{V}}(e) = v}{t(\{l \mid \texttt{return}\ e\}; \{l' \mid \texttt{block}(y); sr\}; \rho)\ o(a,\delta) \to t(\{l' \mid y = v; sr\}; \rho)\ o(a,\delta')}$$

(End)

$$\frac{l^{\mathcal{V}}(\texttt{this}) = o \quad \delta(t) \quad \delta' = \mathit{dec}_t(\delta)}{t(\{l \mid \texttt{return e}\}; \texttt{idle})\ o(a,\delta) \to o(a,\delta')}$$

(Acquire1)

$$\frac{\delta(t) \quad l^{\mathcal{V}}(\texttt{this}) = o \quad (a+l)^{\mathcal{V}}(y) = g \quad (v : J) \in \mathit{export} - (a+l)^{\mathcal{V}}(\bar{x}) \quad J \preceq I}{t(\{l \mid \texttt{try}\ x = \texttt{acquire}\ I\ \texttt{in}\ y\ \texttt{except}\ \bar{x}\ \{s_1\}\ \texttt{else}\ \{s_2\}; sr\}; \rho)\ o(a,\delta)\ g(\mathit{export}) \to t(\{l \mid x = v; s_1; sr\}; \rho)\ o(a,\delta)\ g(\mathit{export})}$$

(Acquire2)

$$\frac{\delta(t) \quad l^{\mathcal{V}}(\texttt{this}) = o \quad (a+l)^{\mathcal{V}}(y) = g \quad \mathit{export}' = \mathit{export} - (a+l)^{\mathcal{V}}(\bar{x}) \quad \mathit{export}' \not\preceq I}{t(\{l \mid \texttt{try}\ x = \texttt{acquire}\ I\ \texttt{in}\ y\ \texttt{except}\ \bar{x}\ \{s_1\}\ \texttt{else}\ \{s_2\}; sr\}; \rho)\ o(a,\delta)\ g(\mathit{export}) \to t(\{l \mid s_2; sr\}; \rho)\ o(a,\delta)\ g(\mathit{export})}$$

(Leave1)

$$\frac{(a+l)^{\mathcal{V}}(y) = g \quad (a+l)^{\mathcal{V}}(x) = v \quad \delta(t) \quad \mathit{export}' = \mathit{export} \setminus \bigcup_{I \in \overline{I}} \{v : I\} \quad l^{\mathcal{V}}(\texttt{this}) = o \quad \mathit{export}' \preceq \overline{I}}{t(\{l \mid \texttt{try}\ x\ \texttt{leaves}\ y\ \texttt{as}\ \overline{I}\ \{s_1\}\ \texttt{else}\ \{s_2\}; sr\}; \rho)\ o(a,\delta)\ g(\mathit{export}) \to t(\{l \mid s_1; sr\}; \rho)\ o(a,\delta)\ g(\mathit{export}')}$$

(Leave2)

$$\frac{(a+l)^{\mathcal{V}}(y) = g \quad (a+l)^{\mathcal{V}}(x) = v \quad \delta(t) \quad \mathit{export}' = \mathit{export} \setminus \bigcup_{I \in \overline{I}} \{v : I\} \quad l^{\mathcal{V}}(\texttt{this}) = o \quad \mathit{export}' \not\preceq \overline{I}}{t(\{l \mid \texttt{try}\ x\ \texttt{leaves}\ y\ \texttt{as}\ \overline{I}\ \{s_1\}\ \texttt{else}\ \{s_2\}; sr\}; \rho)\ o(a,\delta)\ g(\mathit{export}) \to t(\{l \mid s_2; sr\}; \rho)\ o(a,\delta)\ g(\mathit{export})}$$

(Query1)

$$\frac{\delta(t) \quad l^{\mathcal{V}}(\texttt{this}) = o \quad l(z) = (\texttt{Group}(S), g) \quad \mathit{export} \preceq I}{t(\{l \mid \texttt{try}\ z\ \texttt{subtypeOf}\ I\ \{s_1\}\ \texttt{else}\ \{s_2\}; sr\}; \rho)\ g(\mathit{export})\ o(a,\delta) \to t(\{l[z \mapsto (\texttt{Group}(S \cup \{I\}, g))] \mid s_1; sr\}; \rho)\ o(a,\delta)\ g(\mathit{export})}$$

(Query2)

$$\frac{\delta(t) \quad l^{\mathcal{V}}(\texttt{this}) = o \quad l^{\mathcal{V}}(z) = g \quad \mathit{export} \not\preceq I}{t(\{l \mid \texttt{try}\ z\ \texttt{subtypeOf}\ I\ \{s_1\}\ \texttt{else}\ \{s_2\}; sr\}; \rho)\ o(a,\delta)\ g(\mathit{export}) \to t(\{l \mid s_2; sr\}; \rho)\ o(a,\delta)\ g(\mathit{export})}$$

Figure 7: The operational semantics (2).

information of that group. This is crucial in the rules JOIN and LEAVE1, which may actually modify the interface information of the group.

The SKIP rule is standard; being independent of the object, it expresses that a skip has no effect on the object state. The two rules COND1 and COND2 handle the two cases of the conditional statement by evaluating the condition in the object state. The effect of assignment is described by two rules, ASSIGN1 for the assignment to local variables, updating l, and ASSIGN2 for the assignment to fields, updating a. A loop unwinds into a conditional, by rule WHILE. In rule NEW-GROUP, a globally unique group identifier is ensured by the auxiliary predicate $\mathit{fresh}(g)$. An empty group with this identifier is added to the configuration. Rule NEW-THREAD similarly adds a new thread to the configuration, with a globally unique identifier. The new thread gets a stack which consists of a single process frame, corresponding to the called method.

The **new** statement is handled by the NEW-OBJECT rule, where $\mathit{fresh}_C(o')$ and $\mathit{fresh}(t')$ assert that o' and t' do not previously occur in the global configuration and that $\mathit{classOf}(o') = C$. An object with this name is created. The auxiliary function $\mathit{atts}(C, \overline{v})$ maps the declared fields of C to their declared types and default values, and class parameters to declared types and actual values. (The default value of an interface or group type is `null`.) The auxiliary function $\mathit{init}(C, o')$ returns the process corresponding to the init-block of C, binding `this` to type C and value o'. This process is executed in a new thread with identity t', which holds the lock to o'.

Method calls are handled by rule CALL1 for calls to objects, rule CALL2 for releasing calls. and rule CALL3 for calls to groups. We assume that the function type ω of the call has been added during static analysis, as explained in Section 3. Let the auxiliary function $\mathit{bind}(m_\omega, o, C, \overline{v})$ return the process resulting from the activation of a method m of class C in object o such that the function type of m is equal or better than ω.

$$\mathit{bind}(m_\omega, o, C, \overline{v}) \quad = \quad \texttt{error} \quad \textbf{if } C \not\preceq m_\omega$$

The function $\mathit{bind}(m_\omega, o, C, \overline{v})$ is `error` unless C defines a method $T\ m(\overline{T}\ \overline{z})\ \{\overline{T}'\ \overline{z}'; sr;\}$ such that $\overline{T} \to T \preceq \omega$; and in this case we define

$$\mathit{bind}(m_\omega, o, C, \overline{v}) = \{[\overline{z} \mapsto (\overline{T}, \overline{v}), \overline{z}' \mapsto (\overline{T}', \texttt{default}(\overline{T}')), \texttt{this} \mapsto (C, o)] | \texttt{lock}(1); sr;\}$$

The local state maps the parameters of m to their declared types and values $\overline{v}$, the local variables to their declared types and default values, and `this` to

C and o. The *bind* function is deterministic if each class has distinct names for methods with a given number of arguments.

For simplicity, we assume that all methods are *synchronized* in the sense of Java and let the method body in the method activation be preceded by a runtime statement `lock`(n); i.e., the actual method body can only be executed once the thread has taken the lock to the object o n times. In rule Lock-Object a thread t takes the lock δ of an object n times by means of $inc_t(n, \delta)$. This operation succeeds if the lock is either *free* or already taken by t. When a call is made to an object in Call1, the new process $bind(m_\omega, o, C, (a + l)^{\mathcal{V}}(\overline{e}))$ is added to the stack of the thread, where C is the class of the callee. Rule Call2 implements the release mechanism associated with **yield**, which allows a thread to make a method call without keeping the lock of the caller object. The caller remains on the stack, and the thread must compete for the lock before it can execute when control returns. Although the thread may have taken the lock n times, the lock is `free` when the thread leaves the object, and the lock is taken n times upon return. In Call3, a call to a group is reduced to a call to a group or an object *inside* the callee which exports an appropriate interface to the original group in the sense that the called method is supported by the interface. The rules Call1 and Call3 are selected depending on the actual value of the callee y, which may be either an object or a group. Rule Return handles returns from method calls. Here the `block`(y) statement in the frame below the active frame (on the top of the stack) is *replaced* by $y = v$, assigning the value returned from the active frame to y, and the active frame is removed from the stack. Rule Return decrements the lock δ once by means of $dec_t(\delta)$. This operation succeeds if the lock counter is positive, if the lock counter is 1 then the decrement makes the lock `free`. Rule End is similar to rule Return, handling the completion of initialization blocks, main programs, and spawned threads. The associated thread is terminated, in which case the thread no longer holds any locks. This rule takes care of the garbage collection of threads.

The rule Join dynamically extends a group z with support for the services described in the interfaces $\overline{I}$. From the perspective of the thread, the type of the variable referencing the group is extended. From the group's perspective, the *export* set is extended. Observe that in the **join** statement, x may itself be a group; well-typedness ensures that x offers at least $\overline{I}$. Service discovery is handled by the Acquire rules. In Acquire1 the **acquire** statement is replaced by a value v, which is an object or group identifier satisfying the **in** and **except** clauses. The notation $export \preceq I$ means that the *export* list

has an interface J such that $J \preceq I$. If this condition is not satisfied the **else** branch is taken as defined in ACQUIRE2. An **acquire** statement which is not restricted to searching a specific group could be added, but requires constructing a "global" group, e.g., $global(\cup_i\ export_i)$ where $g_i(export_i)$ are all the groups in the configuration.

The **leaves** statement is handled by the rules LEAVE1 for a successful leave and LEAVE2 for an unsuccessful one. A group or object x succeeds in leaving a group if the group continues to provide the same interface support without x. To determine this, we use the subtype relation lifted to group export lists as defined in Figure 5. An entry is redundant if a subtype of the entry is present in the set. The type of the group does not change by a **leaves** statement and hence the object does not need to update information about the group. The branches s_1 or s_2 are chosen depending on whether the interfaces remain supported. The rules QUERY1 and QUERY2 handle the branching determined by a query. If the local group z exports a given interface, the query succeeds and the s_1 branch is taken, updating z with the new interface information. If the query fails the s_2 branch is chosen by QUERY2.

The initial configuration. For a program $P = \overline{IF}\ \overline{CL}\ \{\overline{T}\ \overline{z}; sr\}$, we define the initial configuration to be $o(\varepsilon, (t, 1))\ t(\{[\overline{z} \mapsto (\overline{T}, \texttt{default}(\overline{T})), \texttt{this} \mapsto (C, o)] \mid sr; \}; \texttt{idle})$ where o, t, and C are fresh names such that $classOf(o)$ is C and $\mathsf{Null} \prec C \prec \mathsf{Any}$. The fresh name C plays the role of a *Main* class.

Remarks. Our semantics covers runtime errors resulting from method binding. Runtime errors caused by null pointers are indirectly captured in the sense that execution of the process stops because no more rules can be applied to it. This is the case when a call is made to a null object or group, when joining, leaving, or querying a null group, and when acquiring an interface in a null group. We could add rules explicitly generating the `error` process in these cases; however, as these kinds of errors do not play a role in the results on type safety, we do not need them for our purposes. Static checks ensuring non-null pointer values would be a way to solve these kinds of errors (our semantics guarantees that **this** is not null).

The configurations defined by the operational semantics have state information for each object and thread, and these states include explicit type information. As for the static type system, the types of fields do not change, and the types of local variables that are not group variables do not change. The basic statements have the same effect on the types as in the static type

system. The typing effect of `if`, `try`, and `while` statements is different from that of the static type system, as it uses the typing effect from the chosen branch, whereas the static typing of `if` and `try` statements uses weakening, and the static typing of `while` ignores the effect of the loop. As a consequence the types at runtime are better than (or equal to) than the corresponding types defined by the static type system, which again are better than (or equal to) the declared types.

Notice that all lock handling is managed by the operational semantics, letting lock statements be inserted by *bind* and the rule for **yield**, and letting **return** statements decrease the lock. The initial block of a thread takes the lock of **this** object and releases it by rule End. Assuming we start from an initial configuration, the lock of **this** object is continuously held by the thread, and with the same lock value. Under this assumption, the premise $\delta(t)$ is redundant in all rules except the rule for lock statements.

5. Type Safety

This section extends the type system of Section 3 to runtime configurations and shows type preservation for the execution of well-typed programs.

5.1. Well-Typed Configurations

The extension of the type system to runtime configurations is given in Figure 8. The typing context Γ contains the types of all constant values (including object, group, and thread identities) at runtime. By RTT-Config, a configuration is well-typed if all objects, groups, and threads are well-typed. By RTT-Group, a group is well-typed if all the objects which export interfaces through the group implement these interfaces and that each interface supported by the group according to the type system is exported by an object in the group (checked by RTT-Exps and RTT-Exp). We denote by $CT(C)$ the typing context which maps the fields of C to their declared types. By RTT-Object, an object o is well-typed if its fields a are well-typed in $\Gamma + CT(\Gamma(o))$ and its lock is well-typed. This means that the typing of fields is invariant over execution. Substitutions (the state of fields and local variables) are checked by RTT-Subs and RTT-Sub. By RTT-ObjectLock, the object's lock is well-typed if a thread t holds the lock n times, where n is an integer.

A thread is well-typed by RTT-Thread1 if its stack is well-typed and by RTT-Thread2 if it is idle. A stack is well-typed by RTT-Idle if it is `idle`

and by RTT-Stack if all its processes are well-typed by RTT-Frame1 and RTT-Frame2; i.e., the state of local variables, the `block` statement if the stack is not active, and the method body sr are well-typed. The runtime syntax enforces the usage of the `block` statement only once in a frame, at the head of the frame. Observe that due to the query-mechanism of the language, the types of local program variables in two processes which stem from activations of the same method, may differ at runtime. This is in contrast to the typing of fields. For this reason, the typing context used for typing runtime configurations cannot rely on the statically declared types of program variables. This explains why RTT-Frame1 and RTT-Frame2 extend Γ with the *locally stored typing information* $l^{\mathcal{T}}$ to type check $l^{\mathcal{V}}$, sr, and $\texttt{block}(x)$. The effects of the static type system are not needed here, as they are reflected by how the operational semantics updates this local type information. The additional runtime statement $\texttt{lock}(n)$ is well-typed by RTT-Lock. The rules from the static type checking are reused as appropriate. The remaining rules RTT-Empty, RTT-Free, RTT-Def, and RTT-EmptyGroup are straightforward.

5.2. Subject Reduction

The type system guarantees that the type of *fields* in an object never changes at runtime. The static typing of methods in well-typed programs allows us to establish as Lemma 1 that method binding, if successful, results in a well-typed process at runtime. To show that the `error` process cannot occur in the execution of well-typed programs, it suffices to show that substitutions are always well-typed. Lemma 2 shows that this is the case for the initial configuration and Lemma 3 shows that one execution step preserves runtime well-typedness. Together, these lemmas establish a subject reduction theorem for the language, expressing that well-typedness is preserved during the execution of well-typed programs and in particular that method binding always succeeds. Here, $\rightarrow^*$ denotes the reflexive and transitive closure of the reduction relation $\rightarrow$.

Lemma 1. *Let $m_{\overline{T}\rightarrow T} \in C$ be declared in a well-typed program and let o be an object of that program such that $classOf(o) = C$. If $\Gamma \vdash \overline{v} : \overline{T}'$, $\overline{T}' \preceq \overline{T}$, and $T \preceq T'$, then $\Gamma + CT(C) \vdash bind(m_{\overline{T'}\rightarrow T'}, o, C, \overline{v}) : T'$.*

Lemma 2. *Let P be a program such that $\Gamma \vdash P$ `ok` and let cn be the initial configuration of P. Then there is a Γ' such that $\Gamma' \vdash cn$ `ok` and $\Gamma \subseteq \Gamma'$.*

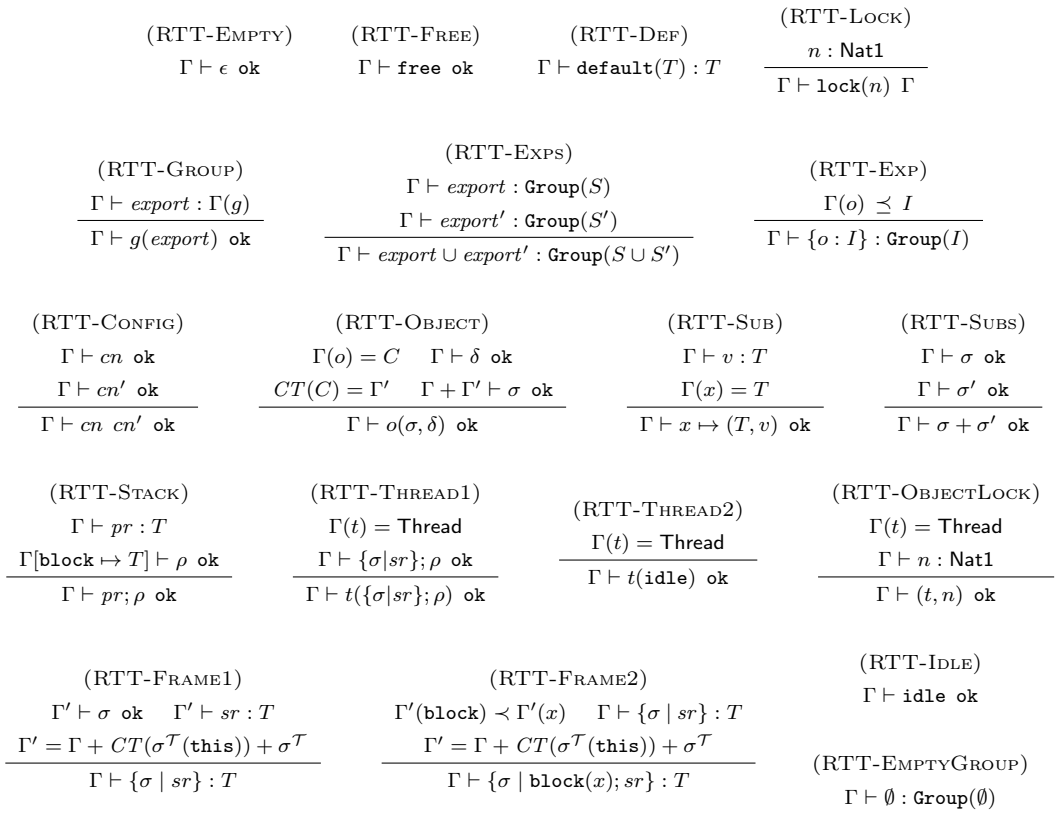

Figure 8: The runtime type system. Nat1 denotes the natural numbers greater than zero.

Lemma 3. *If* $\Gamma \vdash cn$ `ok` *and* $cn \to cn'$ *then there is a* Γ' *such that* $\Gamma' \vdash cn'$ `ok` *and* $\Gamma \subseteq \Gamma'$.

The proofs of the lemmas are found in Appendix A.

Theorem 1 (Subject reduction)**.** *Let* $\Gamma \vdash P$ `ok` *and let* cn *be the initial runtime configuration of* P. *If* $cn \to^* cn'$ *then there is a* Γ' *such that* $\Gamma' \vdash cn'$ `ok` *and* $\Gamma \subseteq \Gamma'$.

Proof. The proof is by induction over the length of the reduction sequence. The two cases follow directly from Lemmas 2 and 3. □

Notice that unsuccessful method binding gives the special process `error`, which is not well-typed since there is no type rule for `error`. Thus our notion of subject reduction implies that `error` may not occur, i.e., method binding

will succeed. However, the execution of a thread may stop if the thread is blocked, in the sense that it does not have the lock to the object on top of the process stack, or is accessing a null pointer. A statement s is said to be *null-free* in a given state if no variable in the statement has the value `null` apart from left-hand-side variables and actual parameters, ignoring inner blocks. A list of statements sr is null-free if the first statement (which may be the **return** statement) is null-free. For example, a **try** statement is null-free if the trial is null-free. We prove a notion of *local progress*, expressing that a thread can proceed when it is not blocked and the next statement is null-free.

Lemma 4 (Local progress)**.** *Consider a well-typed configuration containing a thread* $t(\{l \mid sr\}; \rho)$ *and an object* $o(\sigma, \delta)$*, such that* $l^{\mathcal{V}}(\texttt{this}) = o$ *and* $\delta(t)$*. If* sr *is null-free in the state* $(\sigma + l)$*, there will be exactly one applicable operational rule modifying the thread.*

Together, Theorem 1 and Lemma 4 ensure that "well-typed programs do not cause method-not-understood errors at runtime" in the sense of [14]. Our language deals with concurrent execution and deadlocks may occur. The type system does not restrict deadlocks, but ensures that each non-blocked process in a well-typed configuration can make a transition, and in particular, method binding will succeed, provided the callee is not `null`. The condition of null-freeness may be extended to cover well-defined expressions, i.e., the absence of errors in expressions such as division by zero, and then the progress property would assume well-defined expressions. (Alternatively an exception mechanism could be added to the language.)

6. Diversity in Object Groups

The kernel language formalized in this paper proposes a lightweight notion of dynamic object groups in which the main emphasis is on how service-oriented abstractions combine with groups to allow service discovery, migrating a service from one service provider to another, etc. In our model, objects seen as service providers may offer their services through several groups. The proposed mechanisms are lightweight in that the required additional machinery to handle the groups is very small.

Object groups in object-oriented systems are not a uniform concept, but have been used for a variety of purposes. In this section, we consider an additional construct for communication in object groups and discuss how these constructs fit with different ways of using object groups. For simplicity, we

have not opted for integrating the additional construct in the kernel language. However, the extension of the kernel language is fairly straightforward and does not lead to complications from the typing perspective. For convenience, variable names in the examples will be kept disjoint and we refer to fields without dot-notation; e.g., we write publicGroup rather than **this**.publicGroup.

6.1. Unicast vs. Broadcast

The dynamic object group model considered in the kernel language of this paper is based on *unicast* communication following the standard call and return structure of object-oriented languages. A different notion of dynamic object group can be obtained by considering *broadcast* communication. In such a scenario, all objects in a group who provide the service, receive the call from the group's client. Our model can be adapted to such a scenario, for example by introducing an explicit primitive into the language to express broadcast communication: **broadcast** $y.m(\overline{e})$. Dynamic object groups allow concurrent activities between their objects, so in the multithread setting broadcast is most naturally captured by spawning new threads for the calls to each object. The main issue here is how to handle the reply values from multiple calls. An interesting solution is to collect the replies from the group into a list: if the return type from a method m is type T then the return type to a broadcast to method m would be List<T>. If we assume that the compiler introduces an auxiliary method $\texttt{broadcast}_{m_\omega}$ for every broadcasted method m_ω, this would lead to the following rule in the semantics:

$$\frac{\begin{array}{c}(\textsc{Broadcast})\\ l^{\mathcal{V}}(\texttt{this}) = o \quad \delta(t) \quad (a+l)^{\mathcal{V}}(y) = g\\ S = \{o' | o' : J \in \mathit{export} \wedge J \preceq m_{\overline{T}\to T}\} \quad (a+l)^{\mathcal{T}}(x) = \mathsf{List}\langle T\rangle\end{array}}{\begin{array}{l}o(a,\delta)\ \ t(\{l \mid x = \texttt{broadcast}\ \ y.m_{\overline{T}\to T}(\overline{e}); sr\}; \rho)\ \ g(\mathit{export})\\ \to o(a,\delta)\ \ t(\{l \mid x = \texttt{broadcast}_{m_{\overline{T}\to T}}(S,\overline{e}); sr\}; \rho)\ \ g(\mathit{export})\end{array}}$$

The auxiliary method $\texttt{broadcast}_{m_\omega}$ unwraps method calls to m to a set S of receivers:

```
List<T> broadcast_{m_ω}(Set<J> S, T_1 x_1, ..., T_n x_n){
  T tmp; J o; List<T> replies;
  if S==∅ {
    replies = Nil;
  } else {
    o = some(S);
    tmp = yield o.m(x̄);
```

```
    replies = broadcast_{m_ω}(S\{o}, x_1, ..., x_n);
    replies = Cons(replies,tmp);
  };
  return replies;
}
```

Here, some(S) denotes some value in the set S, Nil denotes the empty list, and Cons(l,e) denotes the list constructor which appends an element e to the list l. The auxiliary method uses **yield** to invoke concurrent execution of m for the different members of S. Observe that with this approach to broadcast, one cannot access a partial response from the group as the replies to all method calls need to be returned before the broadcast succeeds. This restriction would be naturally circumvented in a concurrent object setting with futures (e.g., [6]) rather than with the standard multithreaded concurrency model.

6.2. Channels

Groups can be seen as channels, reminiscent of stubs for remote method calls (RMI); the type system of our kernel language guarantees that the service described by the type of the group is implemented by (at least) one object inside the group. A channel provides an abstraction of who is at either end, but enables communication. There may be more than one sender, and more than one possible receiver. The client makes calls to the group, the receiver is in the group. The group provides anonymity for the receiver end of the channel. The example below illustrates how the receiver in a channel can be dynamically replaced in a way which is transparent to the client, using the kernel language of this paper.

```
Void replace(Service receiver1, Service receiver2, Group(Service) channel){
  receiver2 joins channel as Service;
  try receiver1 leaves channel as Service { skip; } else { skip; };
  return void;
}
```

Channel names are here represented by interface names, and the receiver identity is hidden and may even change dynamically. The two communication models for group communication discussed above, result in different channel behavior. With unicast, only one server will receive the call. With broadcast, all servers will receive the call.

6.3. Roles

One way of managing groups is through different roles, which define available services without specifying the specific implementation of these services. In a type system, roles can be captured through nominal types. Objects may adopt roles or be assigned roles, which associate different nominal types with the object. Roles can be managed at the group level, so an object may have one role in one group and a different role in a different group. Lea [1] distinguishes public and private roles in a group, and relates these to the public and private parts of an object.

In the model of this paper, roles are naturally captured by means of interfaces. These may be dynamically associated with groups. Hence, a group which distinguishes private and public interfaces may be modeled in the kernel language by means of two groups, one which exports public interfaces and another which exports private (group-level) interfaces. In particular, inner groups provide a means to distinguish public calls from intra-group calls. By adding different interfaces to the two groups, an object may provide different services to the group's clients and to the group's members.

The example below illustrates how policies for group membership can be achieved in the kernel language of this paper. In the example, we see how a *group administrator* may create a group with an inner group which functions as a private channel for intra-object communication. In this structure, the inner group has different interfaces from the public group. The two groups are linked via the GroupManager.

```
class GroupManager(Service1 o1, Service2 o2) implements Administrator {
  Group(Service1) publicGroup;
  Group(Administrator,Service2) privateGroup;

  { // Initialization block
    Group(∅) g1; Group(∅) g2;
    g1 = newgroup; g2 = newgroup;
    o1 joins g1 as Service1; publicGroup = g1;
    o2 joins g2 as Service2;
    this joins g2 as Administrator;
    privateGroup = g2;
    g2 joins g1 as Any; return void;
  }

  Group(Administrator,Service1,Service2) joinGroup(Service1 o){
    Group(∅) g1; Group(Administrator,Service2) g2;
    g1 = publicGroup; g2 = privateGroup;
```

```
      o joins g1 as Service1;
      o joins g2 as Service1;
      return g2;
    } ...
}
```

Whereas publicGroup only makes the methods of the Service1 interface available to clients, the inner group has objects providing the interfaces Administrator, Service1, and Service2. Any object which joins the group publicGroup gets the identity of the inner group privateGroup and may use the inner group for intra-group communication. By modifying the joinGroup method such that the state of the GroupManager determines whether the caller may join the group, the GroupManager can seal a group to prevent new members.

6.4. Group Variants

The following list illustrates the diversity of group usage in object-oriented systems [1]. We relate these to the proposed kernel language and to the discussion in Section 6.1 of unicast and broadcast communication in groups.

- *Subscription Groups*: Groups where all calls from clients are broadcast to all group members, without imposing a restriction on how calls are handled. This can be modeled in our setting using the broadcast model discussed in Section 6.1 and a group manager that only accepts objects as members if they implement the particular interface.

- *Work Groups*: Groups where the different members provide different interfaces, such that work tasks and subtasks can be distributed among the members of the group. This is directly supported by unicast in the kernel language.

- *Service Groups*: Groups where any member can handle a call from a client. These are directly supported by unicast in the kernel language where all group members implement the same interface, and similar to subscription groups except that they use unicast instead of broadcast.

- *Resource Groups*: Groups of functionally identical services that may be acquired and released by clients. These are unicast groups, similar to subscription and service groups. They differ in that the intention is that members of a resource group are held by the respective client while the client is using the resource. In the approach of this paper,

resources would leave the group when acquired by a client and rejoin the group when released.

- *Access Groups*: Groups where the members have special privileges. These groups may be either based on unicast or broadcast communication. Access groups based on unicast are directly supported by the kernel language, as illustrated by the GroupManager example which grants group members access to the internal group.

- *Replication Groups*: Groups where all members receive the request, typically to ensure fault tolerance. Replication groups are based on broadcast, but they may require that a reply is returned even if some members fail. Replication groups can be achieved by adapting the broadcast mechanism of Section 6.1 to use, e.g., a list of future variables or a shared return variable instead of a list.

- *Transaction Groups*: Groups where the members follow a particular protocol. These are unicast groups and may in principle be encoded in the kernel language by a group manager for an internal group. Transaction groups are not particularly supported by the approach taken in this paper.

- *Property Groups*: Groups where objects are added or removed based on a particular property (for example a location). In the kernel language, this would require a proactive group controller which monitors objects for the particular property. Property groups are not particularly supported by the approach taken in this paper.

7. Related Work

Object orientation is well-suited for designing small units that encapsulate state with behavior, but it does not directly address the organization of more complex software units with rich interfaces. Two approaches to building flexible and adaptive complex software systems involve, independently, object groups and service discovery. Two main uses of groups appear in the literature: *group communication* and *groups as components*. In the first case, a group is used to facilitate one-to-many communication, and to provide support for, e.g., load balancing. Members of such groups tend to offer the same interface. In the second case, the members of a group typically offer different,

complimentary interfaces and the group acts as a means for composition of objects beyond what standard classes and objects offer. Service discovery allows the binding of a client and a service object via an interface, rather than requiring that the two objects know each others' identity. Our work unifies these approaches in a formal, type-safe setting.

The most common use of object groups is to provide replicated services in order to offer better fault tolerance. Communication to members of a group is via multicast. This idea originated in the Amoeba operating system [15]. The component model Jgroup/ARM [16] adopts this idea to provide autonomous replication management using distributed object groups. In this setting, members of a group maintain a replicated state for reasons of consistency. The ProActive active object programming model [17] supports abstractions for object groups, which enable group communication—via method calls—and various means for synchronizing on the results of such method calls. ProActive is formalized in Caromel and Henrio's Theory of Distributed Objects [7]. A core difference with our work is that ProActive's groups lacks a notion of service discovery.

Another early work on groups is ActorSpaces [18], which combine Actors with Linda's pattern matching facility, allowing both one-to-one communication, multicast, and querying. Unlike our approach, groups in ActorSpaces are intensional: all actors with the same interface belong to the same group. No formalisation is given, nor is typing discussed.

Object groups have been investigated as a modularization unit for objects which is complementary to components. Groups meet the needs of organizing and describing the statics and dynamics of networks of collaborating objects [1]; groups can have many threads of control, they support roles (or interfaces), and objects may dynamically join and leave groups. Lea [1] presents a number of common usages for groups and discusses their design possibilities, inspired from CORBA. Groups have been used to provide an abstraction akin to a notion of component. For example, in Oracle Siebel 8.2 [19], groups are used as units of deployment, units of monitoring, and units of control when deploying and operating components on Siebel servers. Our approach abstracts from most of these details, though groups are treated as first class entities in our calculus.

ProActive's components [7] are similar to our notion of group in that they consist of a collection of active objects working together, that objects can be accessed via interface and that communication can be performed in a one-to-one or many-to-one fashion. One key difference is that ProActive

relies on client and server ports to connect components, whereas our model lacks ports and instead regular interfaces and service discovery are used to connect objects.

Object groups have further been used for coordination purposes. For example, CoLaS [20] is a coordination model based on groups in which objects may join and leave groups. CoLaS goes beyond the model in our paper by allowing very intrusive coordination of message delivery based on a coordinator state. In our model, the groups do not have any state beyond the state of their objects. Similar to our model, objects enroll to group roles (similar to our interfaces). However, unlike our model, objects may leave a group at any time and the coordinator may access the state of participants. The model is implemented in Smalltalk and neither formalization nor typing are discussed [20].

Concurrent object groups have been proposed to define collaborating objects with a single thread of control in programming and modeling languages [21, 22]. In contrast to our dynamic object groups, these concurrent object groups do not have identity and function as runtime restrictions on concurrency rather than as a linguistic concept.

Microsoft's Component Object Model (COM) allows querying a component to check whether it supports a specific interface, similar to the query-mechanism considered in this paper. A component in COM may also have several interfaces, which are independent of each other. In contrast to the model presented in our paper, COM is not object-oriented and the interfaces of a component are stable (i.e., they do not change). COM has proven difficult (or perhaps, uninteresting) to formalize. Pucella develops λ^{COM} [8], a typed λ-calculus which addresses COM components in terms of their interfaces, and discusses extensions to the calculus to capture subtyping, querying for interfaces, and aggregation.

A wide range of service discovery mechanisms exist [23]. The programming language AmbientTalk [24] has built-in service discovery mechanisms, integrated in an object-oriented language with asynchronous method calls and futures. In contrast to our work, AmbientTalk is an untyped language, and lacks any compile time guarantees. Various works formalise the notion of service discovery [25], but they often do so in a formalism quite far removed from the standard setting in which a program using service discovery would be written, namely, an object-oriented setting. For example, Fiadeiro et al.'s model of service discovery and binding takes an algebraic and graph-theoretic approach [26], but it lacks the concise operational notion of service discovery

formalized in our model. No type system is presented either.

Some systems work has been done that combines groups and service discovery mechanisms, such as group-based service discovery mechanisms in mobile ad-hoc networks [27, 28]. In a sense our approach provides language-based abstractions for such a mechanism, except that ours is also tied to interface types to ensure type soundness and includes a notion of exclusion to filter matched services.

Our earlier work [29] enabled objects to advertise and retract interfaces to which other objects could bind, using a primitive service discovery mechanism. A group mechanism was also investigated as a way of providing structure to the services. In that work services were equated with single objects, whereas in the present work a group service is a collection of objects exporting their interfaces. In particular, this means that the type of a group can change over time as it comes to support more functionality.

The key differences with most of the discussed works is that the model in this paper remains within the object-oriented approach, multiple groups may implement an advertised service in different ways, and our formalism offers a transparent group-based service discovery mechanism with primitive exclusion policies. Furthermore, our notion of groups has an implicit and dynamically changing interface.

8. Conclusion

This paper has proposed a formal model for adaptive service-oriented systems, based on a notion of object-oriented groups. We develop a kernel object-oriented language in which groups are first-class citizens in the sense that they may play the role of objects; i.e., a reference typed by an interface may refer to an object or to a group. A main advantage is that several objects may be collected into a group, thereby obtaining a rich interface reflecting a complex service, which can be seen as a single object from the outside. Although objects in our language are restricted to executing one method activation at the time, the language itself is multithreaded and a group may serve many clients at the same time due to inner concurrency.

In contrast to objects, our groups may dynamically add support for an increasing number of interfaces. Group formation is dynamic; join and leave primitives allow the migration of services provided by objects and inner groups as well as software upgrade, provided that interfaces are not removed from a group. An object or group may be part of several groups at the same

time. This gives a very flexible notion of group. By combining inner groups and release using **yield**, the language allows complex group behavior to be expressed in a simple and elegant way, including inner encapsulation and intra-object communication, access control, and mechanisms for intercepting and filtering messages.

In this paper, dynamic object groups are combined with service discovery by means of **acquire** and **subtypeOf** primitives. This allows a programmer to discover services in an open and unknown environment or in a known group, and to query interface support of a given object or group. These mechanisms are formalized in a general object-oriented setting, based on experiences from a prototype implementation of the group and service discovery primitives in Maude [13]. The presented model provides expressive mechanisms for adaptive services in the setting of object-oriented programming with modest conceptual additions. We have developed an operational semantics and type and effects system for the kernel language, and shown the soundness of the approach by a proof of type safety.

The combination of features proposed in this paper suggests that our notion of a group can be made into a powerful programming concept. The work presented may be further extended in a number of directions. The overall goal of our work has been to study an integration of service-oriented and object-oriented paradigms based on a formal foundation. In this paper, we have explored an approach to groups in the setting of multithread concurrency. However, the **yield** mechanism proposed for release takes inspiration from asynchronous method calls with implicit futures in the context of cooperatively scheduled concurrent objects [30]. To better support groups based on broadcast communication and different ways to collect and use results from an *unknown* number of objects in the group, it would be a natural extension of our work to introduce asynchronous method calls, futures, and cooperative scheduling into the language (e.g., [6, 30]). It is also interesting to study the integration into the kernel language of more service-oriented concepts such as for example error propagation and handling, as well as high-level group management operations such as group aggregation.

Acknowledgement. We thank the anonymous reviewers for their detailed comments, which contributed to a significant improvement of the paper.

References

[1] D. Lea, Objects in Groups, available at `http://gee.cs.oswego.edu/dl/groups/groups.html`, 1993.

[2] J. Bjørk, D. Clarke, E. B. Johnsen, O. Owe, A Type-Safe Model of Adaptive Object Groups, in: N. Kokash, A. Ravara (Eds.), Proc. 11th Intl. Workshop on Foundations of Coordination Languages and Self Adaptation (FOCLASA'12), vol. 91 of *Electronic Proceedings in Theoretical Computer Science*, 1–15, 2012.

[3] A. Igarashi, B. C. Pierce, P. Wadler, Featherweight Java: a minimal core calculus for Java and GJ, ACM Transactions on Programming Languages and Systems 23 (3) (2001) 396–450.

[4] C. Flanagan, S. N. Freund, Type-based race detection for Java, in: M. S. Lam (Ed.), Proceedings Conference on Programming Language Design and Implementation (PLDI), ACM, 219–232, 2000.

[5] J. Östlund, T. Wrigstad, Welterweight Java, in: J. Vitek (Ed.), 48th International Conference on Objects, Models, Components, Patterns (TOOLS), vol. 6141 of *Lecture Notes in Computer Science*, Springer, 97–116, 2010.

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

[7] D. Caromel, L. Henrio, A Theory of Distributed Objects - Asynchrony, Mobility, Groups, Components, Springer, 2005.

[8] R. Pucella, Towards a formalization for COM part I: the primitive calculus, in: M. Ibrahim, S. Matsuoka (Eds.), Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'02), ACM, 331–342, 2002.

[9] J.-P. Talpin, P. Jouvelot, Polymorphic Type, Region and Effect Inference., Journal of Functional Programming 2 (3) (1992) 245–271.

[10] T. Amtoft, H. R. Nielson, F. Nielson, Type and effect systems - behaviours for concurrency, Imperial College Press, 1999.

[11] J. M. Lucassen, D. K. Gifford, Polymorphic effect systems, in: Proceedings of the 15th Symposium on Principles of Programming Languages (POPL'88), ACM Press, 47–57, 1988.

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

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

[14] B. C. Pierce, Types and Programming Languages, The MIT Press, 2002.

[15] M. F. Kaashoek, A. S. Tanenbaum, K. Verstoep, Group communication in Amoeba and its applications, Distributed Systems Engineering 1 (1) (1993) 48–58.

[16] H. Meling, A. Montresor, B. E. Helvik, Ö. Babaoglu, Jgroup/ARM: a distributed object group platform with autonomous replication management, Software: Practice and Experience 38 (9) (2008) 885–923.

[17] L. Baduel, F. Baude, D. Caromel, Efficient, flexible, and typed group communications in Java, in: J. E. Moreira, G. Fox, V. Getov (Eds.), Proc. Joint ACM-ISCOPE Conference on Java Grande, ACM, 28–36, 2002.

[18] G. Agha, C. J. Callsen, ActorSpaces: An Open Distributed Programming Paradigm, in: M. C. Chen, R. Halstead (Eds.), Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), ACM, 23–32, 1993.

[19] Oracle Corporation, Siebel Business Applications Documentation, 2010.

[20] J. C. Cruz, S. Ducasse, A Group Based Approach for Coordinating Active Objects, in: P. Ciancarini, A. L. Wolf (Eds.), Third International Conference on Coordination Languages and Models (COORDINATION'99), vol. 1594 of *Lecture Notes in Computer Science*, Springer, 355–370, 1999.

[21] J. Schäfer, A. Poetzsch-Heffter, JCoBox: Generalizing Active Objects to Concurrent Components, in: T. D'Hondt (Ed.), Proc. European Conference on Object-Oriented Programming (ECOOP 2010), vol. 6183 of *Lecture Notes in Computer Science*, Springer, 275–299, 2010.

[22] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, M. Steffen, ABS: A Core Language for Abstract Behavioral Specification, in: B. Aichernig, F. S. de Boer, M. M. Bonsangue (Eds.), Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010), vol. 6957 of *Lecture Notes in Computer Science*, Springer, 142–164, 2011.

[23] P. Hasselmeyer, On Service Discovery Process Types, in: B. Benatallah, F. Casati, P. Traverso (Eds.), Proceedings of the Third International Conference on Service-Oriented Computing (ICSOC 2005), vol. 3826 of *Lecture Notes in Computer Science*, Springer, 144–156, 2005.

[24] J. Dedecker, T. V. Cutsem, S. Mostinckx, T. D'Hondt, W. D. Meuter, Ambient-Oriented Programming in AmbientTalk, in: D. Thomas (Ed.), Proc. 20th European Conference on Object-Oriented Programming, (ECOOP'06), vol. 4067 of *Lecture Notes in Computer Science*, Springer, 230–254, 2006.

[25] A. Lapadula, R. Pugliese, F. Tiezzi, Service Discovery and Negotiation With COWS, Electronic Notes in Theoretical Computer Science 200 (3) (2008) 133–154.

[26] J. L. Fiadeiro, A. Lopes, L. Bocchi, An abstract model of service discovery and binding, Formal Aspects of Computing 23 (4) (2011) 433–463.

[27] D. Chakraborty, A. Joshi, Y. Yesha, T. W. Finin, GSD: a novel group-based service discovery protocol for MANETS, in: Proceedings of The Fourth IEEE Conference on Mobile and Wireless Communications Networks, IEEE, 140–144, 2002.

[28] Z. Gao, L. Wang, M. Yang, X. Yang, CNPGSDP: An efficient group-based service discovery protocol for MANETs, Computer Networks 50 (16) (2006) 3165–3182.

[29] D. Clarke, E. B. Johnsen, O. Owe, Concurrent Objects à la Carte, in: D. Dams, U. Hannemann, M. Steffen (Eds.), Concurrency, Composition-

ality, and Correctness, vol. 5930 of *Lecture Notes in Computer Science*, Springer, 185–206, 2010.

[30] E. B. Johnsen, O. Owe, An Asynchronous Communication Model for Distributed Concurrent Objects, Software and Systems Modeling 6 (1) (2007) 35–58.

Appendix A. Proof of Type Preservation

This appendix includes proofs of the lemmas from Section 5.2.

Lemma 1. *Let $m_{\overline{T}\to T} \in C$ be declared in a well-typed program and let o be an object of that program such that classOf$(o) = C$. If $\Gamma \vdash \overline{v} : \overline{T}'$, $\overline{T}' \preceq \overline{T}$, and $T \preceq T'$, then $\Gamma + CT(C) \vdash bind(m_{\overline{T'}\to T'}, o, C, \overline{v}) : T'$.*

Proof. Since *classOf*$(o) = C$, o cannot be `null`. Since $m_{\overline{T}\to T} \in C$, $bind(m_{\overline{T}\to T}, o, C, \overline{v})$ must give a non-error process $\{l \mid \texttt{lock}(1); s; \texttt{return}\ e\}$ such that C has the method $T'\ m(\overline{T_z}\ \overline{z})\ \{\overline{T_{z'}}\ \overline{z}'; s; \texttt{return}\ e;\}$ where l is $[\overline{z} \mapsto (\overline{T_z}, \overline{v}), \overline{z}' \mapsto (\overline{T_{z'}}, \texttt{default}(\overline{T_{z'}}))$, $\texttt{this} \mapsto (C, o)]$. So the local typing context is $l^{\mathcal{T}} = [\overline{z} \mapsto \overline{T_z}, \overline{z}' \mapsto \overline{T_{z'}}, \texttt{this} \mapsto C]$. We need to show that $\Gamma \vdash \{l|\texttt{lock}(1); s; \texttt{return}\, e;\} : T$. Let $\Gamma' = \Gamma + CT(C) + l^{\mathcal{T}}$.

We first show $\Gamma' \vdash l$ `ok`. By assumption, $\Gamma \vdash \overline{v} : \overline{T}$, and $\Gamma(o) = C$. Thus $\overline{v} : \overline{T_z}$ since $\overline{T} \preceq \overline{T_z}$. Since $\overline{v}, o$ are values, $\Gamma' \vdash \overline{v} : \overline{T_z}$ and $\Gamma'(o) = C$. Since $\Gamma'(\overline{z}) = \overline{T_z}$ and $\Gamma'(\overline{v}) = \overline{T_z}$, by RTT-Sub and RTT-Subs we have $\Gamma' \vdash \overline{z} \mapsto (\overline{T_z}, \overline{v})$ `ok`. By RTT-Def, $\Gamma' \vdash \texttt{default}(\overline{T_{z'}}) : \overline{T_{z'}}$, so by RTT-Sub and RTT-Subs we have $\Gamma' \vdash \overline{z'} \mapsto (\overline{T_{z'}}, \texttt{default}(\overline{T_{z'}}))$ `ok`. Since $\Gamma'(\texttt{this}) = C$ and $\Gamma'(o) = C$, by RTT-Sub we have $\Gamma' \vdash \texttt{this} \mapsto (C, o)$ `ok`, and by RTT-Subs $\Gamma' \vdash l$ `ok` follows by composition.

We next show $\Gamma' \vdash \texttt{lock}(1); s; \texttt{return}\ e : T$. Since m is well-typed in C we have from T-Class, T-Method, and RTT-Lock that $\Gamma[\texttt{this} \mapsto C] + CT(C) + [\overline{z} \mapsto \overline{T_z}, \overline{z'} \mapsto \overline{T_{z'}}] \vdash \texttt{lock}(1); s; \texttt{return}\ e : T'$. Since $l^{\mathcal{T}} = \overline{z} \mapsto \overline{T_z}, \overline{z'} \mapsto \overline{T_{z'}}, \texttt{this} \mapsto C$, we get $\Gamma + CT(C) + l^{\mathcal{T}} \vdash \texttt{lock}(1); s; \texttt{return}\ e : T'$. By RTT-Frame1 we get $\Gamma \vdash \{l|\texttt{lock}(1); s; \texttt{return}\ e\} : T$, since $T' \preceq T$. □

Lemma 2. *Let P be a program such that $\Gamma \vdash P$ `ok` and let cn be the initial configuration of P. Then there is a Γ' such that $\Gamma' \vdash cn$ `ok` and $\Gamma \subseteq \Gamma'$.*

Proof. Let us assume that $P = \overline{IF}\ \overline{CL}\ \{\overline{T}\ \overline{z}; sr;\}$, then the initial state is $o(\varepsilon, (t, 1))\ t([\{\overline{z} \mapsto (\overline{T}, \texttt{default}(\overline{T})), \texttt{this} \mapsto (C, o)]|sr;\}; \texttt{idle})$ for fresh

names o, t, and C. By assumption, $C \prec \mathsf{Any}$. We assume that $CT(C) = \varepsilon$. Let $\Gamma' = \Gamma[o \mapsto C, t \mapsto \mathsf{Thread}]$.

Obviously, by RTT-Object, $\Gamma' \vdash o(\varepsilon, (t, 1))$ $\mathtt{ok}$. By T-Program, $\Gamma'[\overline{z} \mapsto \overline{T}] \vdash sr : \mathsf{Void}$. By RTT-Def, $\Gamma'[\overline{z} \mapsto \overline{T}] \vdash z_i \mapsto (T_i, \mathtt{default}(T_i))$. By RTT-Sub, $\Gamma'[\mathtt{this} \mapsto (C, o)] \vdash \mathtt{this} \mapsto (C, o)$ $\mathtt{ok}$. Then by RTT-Subs, RTT-Frame1, RTT-Stack, RTT-Thread1, and RTT-Config, $\Gamma' \vdash cn$ $\mathtt{ok}$. □

Lemma 3. *If* $\Gamma \vdash cn$ $\mathtt{ok}$ *and* $cn \to cn'$ *then there is a* Γ' *such that* $\Gamma' \vdash cn'$ $\mathtt{ok}$ *and* $\Gamma \subseteq \Gamma'$.

Proof. The proof is by cases over the reduction rules of the semantics. To help readability, we let σ denote the typing context $CT(\Gamma(\mathtt{this}))$, i.e., the declared types of the fields of the current object. We use the notation $\Gamma \vdash s$ $\mathtt{ok}$ instead of $\Gamma \vdash s\ \Delta$ when the effect Δ is uninteresting.

Case Skip. Let $cn = cn_0\ t(\{l|\mathtt{skip}; sr; \}; \rho)$ and $cn' = cn_0\ t(\{l|sr; \}; \rho)$. By assumption, $\Gamma \vdash cn$ $\mathtt{ok}$, so $\Gamma \vdash cn_0$ $\mathtt{ok}$ and $\Gamma \vdash t(\{l|\mathtt{skip}; sr; \}; \rho)$ $\mathtt{ok}$. Then, by RTT-Thread1, RTT-Stack, RTT-Frame1, RTT-Frame2, and T-Composition, we know that $\Gamma \vdash t(\{l|sr; \}; \rho)$ $\mathtt{ok}$, and $\Gamma \vdash cn'$ $\mathtt{ok}$.

Case While. Let $cn = cn_0\ o(a, \delta)\ t(\{l|\mathtt{while}\, e\, \{s; \}; sr; \}; \rho)$ and $cn' = cn_0\ o(a, \delta)\ t(\{l|\mathtt{if}\, e\, \{s; \mathtt{while}\, e\, \{s; \}\}\, \mathtt{else}\, \{\mathtt{skip}\}; sr; \}; \rho)$. By assumption, $\Gamma \vdash cn$ $\mathtt{ok}$, so $\Gamma + \sigma + l^{\mathcal{T}} \vdash e : \mathsf{Bool}$ and $\Gamma + \sigma + l^{\mathcal{T}} \vdash s$ $\mathtt{ok}$. It follows that $\Gamma + \sigma + l^{\mathcal{T}} \vdash \mathtt{if}\, e\, \{s; \mathtt{while}\, e\, \{s; \}\}\, \mathtt{else}\, \{\mathtt{skip}\}$ $\mathtt{ok}$, and we get that $\Gamma \vdash cn'$ $\mathtt{ok}$.

Case Cond1. Let $cn = cn_0\ o(a, \delta)\ t(\{l|\mathtt{if}\, e\, \{s_1; \}\, \mathtt{else}\, \{s_2; \}; sr; \}; \rho)$ and $cn' = cn_0\ o(a, \delta)\ t(\{l|s_1; sr; \}; \rho)$. By assumption, $\Gamma \vdash cn$ $\mathtt{ok}$, so $\Gamma + \sigma + l^{\mathcal{T}} \vdash s_1$ $\mathtt{ok}$ and $\Gamma + \sigma + l^{\mathcal{T}} \vdash sr$ $\mathtt{ok}$. Then, $\Gamma + \sigma + l^{\mathcal{T}} \vdash s_1; sr$ $\mathtt{ok}$ and it follows that $\Gamma \vdash cn'$ $\mathtt{ok}$.

Case Cond2. Let $cn = cn_0\ o(a, \delta)\ t(\{l|\mathtt{if}\, e\, \{s_1; \}\, \mathtt{else}\, \{s_2; \}; sr; \}; \rho)$ and $cn' = cn_0\ o(a, \delta)\ t(\{l|s_2; sr; \}; \rho)$. By assumption, $\Gamma \vdash cn$ $\mathtt{ok}$, so $\Gamma + \sigma + l^{\mathcal{T}} \vdash s_2$ $\mathtt{ok}$ and $\Gamma + \sigma + l^{\mathcal{T}} \vdash sr$ $\mathtt{ok}$. Then, $\Gamma + \sigma + l^{\mathcal{T}} \vdash s_2; sr$ $\mathtt{ok}$ and it follows that $\Gamma \vdash cn'$ $\mathtt{ok}$.

Case Assign1. Let $cn = cn_0\ o(a, \delta)\ t(\{l|x = e; sr; \}; \rho)$ and $cn' = cn_0\ o(a, \delta)\ t(\{l[x \mapsto (T, v)]|sr; \}; \rho)$, where $l^{\mathcal{T}}(x) = T$ and $(a + l)^{\mathcal{V}}(e) = v$. By assumption, $\Gamma \vdash cn$ $\mathtt{ok}$, so $\Gamma + \sigma + l^{\mathcal{T}} \vdash x = e$ $\mathtt{ok}$. Since $\Gamma \vdash o(a, \delta)$ $\mathtt{ok}$ and $\Gamma \vdash t(\{l|x = e; sr; \}; \rho)$ $\mathtt{ok}$, we know that $\Gamma + \sigma + l^{\mathcal{T}} \vdash v : T$. Then $\Gamma + \sigma + l^{\mathcal{T}} \vdash x \mapsto (T, v)$ $\mathtt{ok}$, so $\Gamma \vdash t(\{l[x \mapsto (T, v)]|sr; \}; \rho)$ $\mathtt{ok}$, and finally $\Gamma \vdash cn'$ $\mathtt{ok}$.

Case Assign2. Let $cn = cn_0\ o(a, \delta)\ t(\{l|x = e; sr; \}; \rho)$ and $cn' = cn_0\ o(a[x \mapsto (T, v)], \delta)\ t(\{l|sr; \}; \rho)$, where $\sigma(x) = T$ and $(a + l)^{\mathcal{V}}(e) = v$. By

assumption, $\Gamma \vdash cn$ $\texttt{ok}$, so $\Gamma + \sigma + l^{\mathcal{T}} \vdash x = e$ $\texttt{ok}$. Since $\Gamma \vdash o(a,\delta)$ $\texttt{ok}$ and $\Gamma \vdash t(\{l|x = e; sr;\}; \rho)$ $\texttt{ok}$, we know that $\Gamma + \sigma + l^{\mathcal{T}} \vdash v : T$. Then $\Gamma + \sigma + l^{\mathcal{T}} \vdash x \mapsto (T, v)$ $\texttt{ok}$, so $\Gamma \vdash o(a[x \mapsto (T,v)], \delta)$ $\texttt{ok}$. Since $\Gamma \vdash t(\{l|sr;\}; \rho)$ $\texttt{ok}$, we get $\Gamma \vdash cn'$ $\texttt{ok}$.

Case LOCK-OBJECT. Let $cn = cn_0\ o(a,\delta)\ t(\{l|\texttt{lock}(n); sr;\}; \rho)$ and $cn' = cn_0\ o(a,\delta')\ t(\{l|sr;\}; \rho)$. By assumption, $\Gamma \vdash cn$ $\texttt{ok}$, so $\Gamma + \sigma \vdash a$ $\texttt{ok}$ and $\Gamma + \sigma + l^{\mathcal{T}} \vdash \texttt{lock}(n); sr : T$ for some T. Since $\Gamma \vdash \delta$ $\texttt{ok}$, we get from RTT-OBJECTLOCK that $\delta' = inc_t(1, \delta) = (t, n)$ for some n, so $\Gamma \vdash \delta'$ $\texttt{ok}$ and it follows that $\Gamma \vdash cn'$ $\texttt{ok}$.

Case NEW-GROUP. Let $cn = cn_0\ o(a,\delta)\ t(\{l|x = \texttt{newgroup}; sr;\}; \rho)$ and $cn' = cn_0\ o(a,\delta)\ t(\{l|x = g; sr;\}; \rho)\ g(\emptyset)$. By assumption, $\Gamma \vdash cn$ $\texttt{ok}$, so $\Gamma + \sigma + l^{\mathcal{T}} \vdash x = \texttt{newgroup}; sr : T$. In particular, by T-ASSIGN, $\Gamma + \sigma + l^{\mathcal{T}} \vdash x : \texttt{Group}(\emptyset)$. Since g is fresh, $g \notin dom(\Gamma)$. Let $\Gamma' = \Gamma[g \mapsto \texttt{Group}(\emptyset)]$. Then $\Gamma' \vdash cn_0\ o(a,\delta)$ $\texttt{ok}$, $\Gamma' + \sigma + l^{\mathcal{T}} \vdash x = g$ $\texttt{ok}$, and $\Gamma' + \sigma + l^{\mathcal{T}} \vdash sr : T$. It follows that $\Gamma' \vdash t(\{l|x = g; sr;\}; \rho)$ $\texttt{ok}$. By RTT-GROUP, $\Gamma' \vdash g(\emptyset)$ $\texttt{ok}$, and it follows that $\Gamma' \vdash cn'$ $\texttt{ok}$.

Case NEW-THREAD. Let $cn = cn_0\ o(a,\delta)\ t(\{l|\texttt{spawn}\, x.m(\overline{e}); sr;\}; \rho)$ and $cn' = cn_0\ o(a,\delta)\ t(\{l|sr;\}; \rho)\ t'(pr; \texttt{idle})$. Let $\Gamma' = \Gamma + \sigma + l^{\mathcal{T}}$ and $\Gamma'' = \Gamma[t' \mapsto \mathsf{Thread}]$. By assumption, $\Gamma + \sigma \vdash \{l|\texttt{spawn}\, x.m(\overline{e}); sr;\} : T$, so $\Gamma + \sigma \vdash \{l|sr;\} : T$ and by T-SPAWN we have $\Gamma'(x) \preceq m_{\Gamma'(\overline{e})} \to Void$. By Lemma 1 we then get $\Gamma \vdash pr : T$, and by RTT-STACK $\Gamma \vdash pr; \texttt{idle}$ $\texttt{ok}$. We have $\Gamma'' \vdash t'(pr; \texttt{idle})$ $\texttt{ok}$ and $\Gamma'' \vdash cn'$ $\texttt{ok}$ follows.

Case NEW-OBJECT. Let $cn = cn_0\ o(a,\delta)\ t(\{l|x = \texttt{new}\ C(\overline{e}); sr;\}; \rho)$ and $cn' = cn_0\ o(a,\delta)\ t(\{l|x = o'; sr;\}; \rho)\ o'(a', (t', 1))\ t'(pr; \texttt{idle})$, where $pr = \{\overline{z} \mapsto (\overline{T}, \texttt{default}(\overline{T})), \texttt{this} \mapsto (C, o')|sr';\}$ where $\overline{z}$ of type $\overline{T}$ are the local variables of the initiator of C. By assumption, $\Gamma \vdash cn$ $\texttt{ok}$, so $\Gamma + \sigma + l^{\mathcal{T}} \vdash x = \texttt{new}\ C(\overline{e}); sr : T$. From T-NEW, we can assume that $\Gamma(x) = I$ such that $C \prec I$, and $\Gamma + \sigma + l^{\mathcal{T}} \vdash \overline{e} : ptypes(C)$. Since o' is fresh, $o' \notin dom(\Gamma)$. Let $\Gamma' = \Gamma[o \mapsto C]$. Then $\Gamma' \vdash cn_0\ o(a,\delta)$ $\texttt{ok}$. We need to show that t, t', and o' are well-typed in Γ'.

It follows from the induction hypothesis that $\Gamma' + \sigma + l^{\mathcal{T}} \vdash x = o'$ $\texttt{ok}$ and consequently $\Gamma' + \sigma + l^{\mathcal{T}} \vdash x = o'; sr : T$. Then, $\Gamma' + \sigma \vdash \{l|x = \texttt{new}\, C(\overline{e}); sr;\} : T$ and t is well-typed in Γ'. Since $\Gamma + \sigma \vdash a$ $\texttt{ok}$ and $\Gamma + \sigma + l^{\mathcal{T}} \vdash l$ $\texttt{ok}$, we have that $\Gamma + \sigma + l^{\mathcal{T}} \vdash (a + l)^{\mathcal{V}}(\overline{e}) : ptypes(C)$. It follows that $\Gamma' \vdash atts(C, (a + l)^{\mathcal{V}}(\overline{e}))$ $\texttt{ok}$ and o' is well-typed in Γ'. Since C is well typed in Γ', we have $\Gamma' + CT(C)[\overline{z} \mapsto \overline{T}] \vdash sr' : \mathsf{Void}$. It follows that $\Gamma' \vdash pr$ $\texttt{ok}$ and the thread t' is well-typed in Γ'.

Case CALL1. Let $cn = cn_0\ o(a,\delta)\ t(\{l|x = y.m_\omega(\overline{e}); sr;\}; \rho)$ and $cn' =$

$cn_0\ o(a,\delta)\ t(pr;\{l|\texttt{block}(x);sr;\};\rho)$. By assumption, $\Gamma \vdash \{l|sr\};\rho\ \texttt{ok}$. Let Γ' denote $\Gamma + \sigma + l^{\mathcal{T}}$ and assume that $\Gamma'(x) = T$. By assumption, $\Gamma \vdash cn\ \texttt{ok}$, so by T-Call we have $\Gamma'(y) \preceq m_{\Gamma'(\overline{e})\to T}$. By Lemma 1 we then get $\Gamma \vdash pr : T$, and it follows from RTT-Stack that $\Gamma \vdash pr;\{l|\texttt{block}(x);sr\};\rho\ \texttt{ok}$.

Case Call2. By RTT-ObjectLock, $\Gamma \vdash (t,n)\ \texttt{ok}$. The case is then similar to *Case* Call1.

Case Call3. Let $cn = cn_0\ o(a,\delta)\ t(\{l|x = y.m_\omega(\overline{e});sr;\};\rho)\ g(export)$ and $cn' = cn_0\ o(a,\delta)\ t(\{l|x = v.m_\omega(\overline{e});sr;\};\rho)\ g(export)$. Let Γ' denote $\Gamma+\sigma+l^{\mathcal{T}}$. By assumption, $\Gamma \vdash cn\ \texttt{ok}$, so by T-Call we have $\Gamma' \vdash x = y.m(\overline{e})\ \texttt{ok}$ where $\Gamma'(y) \preceq m_{\Gamma'(\overline{e})\to\Gamma'(x)}$. Since $v : I \in export$ we know by RTT-Exp that $\Gamma'(v) \preceq I$. Hence, $\Gamma' \vdash v : I$ and $\Gamma' \vdash x = v.m(\overline{e})\ \texttt{ok}$. It follows that $\Gamma \vdash cn'\ \texttt{ok}$.

Case Join. Let $cn = cn_0\ o(a,\delta)\ t(\{l|x\ \texttt{joins}\ z\ \texttt{as}\ \overline{I};sr;\};\rho)\ g(export)$ and $cn' = cn_0\ o(a,\delta)\ t(\{l[z \mapsto (T,g)]|sr;\};\rho)\ g(export')$, where $T = \texttt{Group}(S \cup \overline{I})$. By assumption, $\Gamma \vdash cn\ \texttt{ok}$, so $\Gamma+\sigma+l^{\mathcal{T}} \vdash x\ \texttt{joins}\ z\ \texttt{as}\ \overline{I};sr : T$. Assume that $l(z) = (\texttt{Group}(S),g)$ and that $(a+l)^{\mathcal{V}}(x) = o'$. By T-Join, $(\Gamma+\sigma+l^{\mathcal{T}})(x) \preceq \overline{I}$, and since $\Gamma + \sigma + l^{\mathcal{T}} \vdash l\ \texttt{ok}$, we know that $\Gamma(o') \preceq \overline{I}$. It follows from T-Join and T-Composition that $\Gamma + \sigma + l^{\mathcal{T}}[z \mapsto \texttt{Group}(T)] \vdash sr : T$. Let $export' = \bigcup_{I\in\overline{I}}\{o' : I\} \cup export$. By assumption, $\Gamma \vdash export\ \texttt{ok}$ and since $\Gamma(o') \preceq \overline{I}$ we know by RTT-Exp that $\Gamma \vdash g(export')\ \texttt{ok}$. It follows that $\Gamma \vdash cn'\ \texttt{ok}$.

Case Return. Let $cn = cn_0\ o(a,\delta)\ t(\{l|\texttt{return}\ e\};\{l'|\texttt{block}(y);sr;\};\rho)$ and $cn' = cn_0\ o(a,\delta')\ t(\{l'|y = v;sr;\};\rho)$. By assumption, $\Gamma \vdash cn\ \texttt{ok}$, so by RTT-Stack $\Gamma + \sigma + l^{\mathcal{T}} \vdash e : T$ for some T such that $\Gamma[\texttt{block} \mapsto T] \vdash \{l'|\texttt{block}(y);sr;\};\rho)\ \texttt{ok}$. Since $\Gamma + \sigma \vdash a\ \texttt{ok}$ and $\Gamma + \sigma + l^{\mathcal{T}} \vdash l\ \texttt{ok}$, we have that $\Gamma + \sigma + l^{\mathcal{T}} \vdash v : T$. It follows that $\Gamma \vdash \{l'|y = v;sr;\};\rho)\ \texttt{ok}$. Since $dec_t(\delta)$ is well-defined, either $\delta' = \texttt{free}$, which is well-typed by RTT-Free, or $\delta = (t,n+1)$ which is well-typed by RTT-ObjectLock since $\Gamma(t) = \mathsf{Thread}$ by assumption. It follows that $\Gamma \vdash cn'\ \texttt{ok}$.

Case End. Follows directly from the induction hypothesis.

Case Leave1. Let $cn = cn_0\ o(a,\delta)\ t(\{l|\texttt{try}\ x\ \texttt{leaves}\ y\ \texttt{as}\ \overline{I}\ \{s_1\}\ \texttt{else}\ \{s_2\};sr;\};\rho)\ g(export)$ and $cn' = cn_0\ o(a,\delta)\ t(\{l|s_1;sr;\};\rho)\ g(export')$. By assumption, $\Gamma \vdash cn\ \texttt{ok}$, so $\Gamma \vdash g(export)\ \texttt{ok}$ and $\Gamma+\sigma \vdash \{l|\texttt{try}\ x\ \texttt{leaves}\ y\ \texttt{as}\ \overline{I}\ \{s_1\}\ \texttt{else}\ \{s_2\};sr;\} : T$. It is obvious that $\Gamma + \sigma \vdash \{l|s_1;sr;\} : T$ and that $\Gamma \vdash g(export')\ \texttt{ok}$ where $export' \subseteq export$. Since $export' \preceq \overline{I}$, we know that if $\Gamma \vdash g : \overline{I}$ with $g(export)$ then $\Gamma \vdash g : \overline{I}$ still holds with $g(export')$. It follows that $\Gamma \vdash cn\ g(export')\ \texttt{ok}$.

Case LEAVE2. Follows directly from the induction hypothesis.

Case QUERY1. Let $cn = cn_0\ o(a,\delta)\ t(\{l|\texttt{try}\ z\ \texttt{subtypeOf}\ I\ \{s_1\}\ \texttt{else}\ \{s_2\}; sr;\}; \rho)\ g(export)$ and $cn' = cn_0\ o(a,\delta)\ t(\{l[z \mapsto (\texttt{Group}(S \cup \{I\}, g))]|s_1; sr;\}; \rho)\ g(export)$. By assumption, $\Gamma \vdash cn\ \texttt{ok}$, so $\Gamma + \sigma \vdash \{l|\texttt{try}\ z\ \texttt{subtypeOf}\ I\ \{s_1\}\ \texttt{else}\ \{s_2\}; sr;\} : T$. Let $(l)^{\mathcal{T}}(z) = \texttt{Group}(S)$. By T-INSPECT, we know that $\Gamma + \sigma + l^{\mathcal{T}}[z \mapsto \texttt{Group}(S \cup I)] \vdash s_1\ \texttt{ok}$, and consequently $\Gamma + \sigma + l^{\mathcal{T}}[z \mapsto \texttt{Group}(S \cup I)] \vdash s_1; sr : T$. It follows that $\Gamma \vdash cn'\ \texttt{ok}$.

Case QUERY2. Follows directly from the induction hypothesis.

Case ACQUIRE1. Let $cn = cn_0\ o(a,\delta)\ t(\{l|\texttt{try}\ x = \texttt{acquire}\ I\ \texttt{in}\ y\ \texttt{except}\ \bar{x}; \{s_1\}\ \texttt{else}\ \{s_2\}; sr;\};\ \rho)\ g(export)$ and $cn' = cn_0\ o(a,\delta)\ t(\{l|x = v; s_1; sr;\}; \rho)\ g(export)$, where $(v : J) \in export$. By assumption, $\Gamma \vdash cn\ \texttt{ok}$, so $\Gamma + \sigma \vdash \{l|\texttt{try}\ x = \texttt{acquire}\ I\ \texttt{in}\ y\ \texttt{except}\ \bar{x}; \{s_1\}\ \texttt{else}\ \{s_2\}; sr;\} : T$ and $\Gamma \vdash g((v : J) \cup export)\ \texttt{ok}$. It follows from RTT-EXP that $\Gamma(v) \preceq J$ and since $J \preceq I$, we have $\Gamma + \sigma \vdash \{l|x = v; s_1; sr;\} : T$ and consequently $\Gamma \vdash cn'\ \texttt{ok}$.

Case ACQUIRE2. Follows directly from the induction hypothesis. □

Lemma 4. *Consider a well-typed configuration containing a thread* $t(\{l|sr\}; \rho)$ *and an object* $o(\sigma, \delta)$*, such that* $l^{\mathcal{V}}(\texttt{this}) = o$ *and* $\delta(t)$*. If* sr *is null-free in the state* $(\sigma + l)$*, there will be exactly one applicable operational rule modifying the thread.*

Proof. The proof is by cases over the first statement of the statement list sr, which may be the **return** statement. We may assume that sr is well-typed. In our core language a well-typed expression e can be evaluated to a well-defined value, using $(\sigma + l)^{\mathcal{V}}(e)$. If we were to consider errors in expressions, we would need a strengthened notion of null-free, so that we could assume error-free expressions in the proof.

Case **skip**. When sr starts with **skip**, the SKIP rule can be applied applied since there are no premises on the rule.

Case **while**. When sr starts with a **while** statement, the WHILE rule can be applied applied since there are no premises on the rule.

Case **if** statement. A well-typed Boolean expression (in our core language) will evaluate to true or false. In either case, one rule will apply.

Case z=e. Rule ASSIGN1 applies. (Here null-freeness is not needed.)

Case f=e. Rule ASSIGN2 applies. (Again null-freeness is not needed.)

Case lock(n) statement. The LOCK-OBJECT rule can be applied since $\delta(t)$ implies that $inc_t(n, \delta)$ is well-defined.

Case **newgroup**, **spawn**, and **new** object. The corresponding rule can be applied, given that fresh names (of each category) can be generated. For **spawn**, the callee is not null by the assumption of null-freeness.

Case method call 1: a normal call on an object. By null-freeness, the callee is not null and Rule CALL1 applies.

Case method call 2: a yielding call on an object. By null-freeness, the callee is not null and Rule CALL2 applies.

Case method call 3: a call on a group. By null-freeness, the callee is a non-null group, say $(\texttt{Group}(S), g)$, and by well-typedness of the call, the callee supports an interface I with a method m of the function type ω generated during typing. Since the configuration is well-typed $(\sigma + l)^{\mathcal{V}}(g)$ must contain an object v of type (better than) I. Thus the premises of Rule CALL3 can be satisfied, and the rule can be applied.

Case x **joins** z **as** $\overline{I}$. By null-freeness and well-typedness, z is a non-null group, and rule JOIN applies.

Case **return**. Observe that in a well-typed configuration, a stack consists of an active process at the top of the stack and of (suspended) frames below the active process such that the frame at the bottom of the stack is `idle`. A **return** may only occur as the last statement in the active process or in a frame on the stack. The active frame cannot contain a `block` statement, whereas a frame below the active process must begin with a `block` statement or it is `idle`. Thus, it suffices to consider these two cases for **return** and the rules RETURN or END will apply, respectively. ($\delta(t)$ implies that $dec_t(\delta)$ is defined.)

Case x= **acquire** l **in** y **except** $\overline{x}$. By null-freeness and well-typedness, y will bind to a group. This group may or may not have an object exporting I (other than objects in $\overline{x}$). Rule ACQUIRE1 and rule ACQUIRE2 cover the two cases.

Case x **leaves** y **as** $\overline{I}$. By null-freeness and well-typedness, y will bind to a group. The group may or may not have objects exporting each of $\overline{I}$ (ignoring the object x). Rule LEAVE1 and rule LEAVE2 cover the two cases.

Case z **subtypeOf** l. By null-freeness and well-typedness, z will bind to a group. The group may or may not export I. Rule QUERY1 and rule QUERY2 cover the two cases. □

A.2 Fault Model Design Space for Cooperative Concurrency

Fault Model Design Space for Cooperative Concurrency *

Ivan Lanese[1], Michael Lienhardt[1], Mario Bravetti[1], Einar Broch Johnsen[2], Rudolf Schlatte[2], Volker Stolz[2], and Gianluigi Zavattaro[1]

[1] Focus Team, Università di Bologna/INRIA, Italy
{lanese,lienhard,bravetti,zavattar}@cs.unibo.it
[2] Department of Informatics, University of Oslo, Norway
{einarj,rudi,stolz}@ifi.uio.no

Abstract. This paper critically discusses the different choices that have to be made when defining a fault model for an object-oriented programming language. We consider in particular the ABS language, and analyze the interplay between the fault model and the main features of ABS, namely the cooperative concurrency model, based on asynchronous method invocations whose return results via futures, and its emphasis on static analysis based on invariants.

1 Introduction

General-purpose modeling languages exploit *abstraction* to reduce complexity [20]: modeling is the act of describing a system succinctly by leaving out some aspects of its behavior or structure. Software models primarily focus on the functional behavior and the logical composition of the software. Modeling formalisms can have varying levels of detail and can express structural properties (for example UML diagrams), interactions (π-calculus), or the effects of functions or methods (pre- and post-conditions), etc.

Concurrent and distributed systems demand flexible communication forms between distributed processes. While object-orientation is a natural paradigm for distributed systems [15], the tight coupling between objects traditionally enforced by method calls may be criticized. Concurrent (or active) objects have been proposed as an approach to concurrency that blends naturally with object-oriented programming [1, 22, 32]. Several slightly differently flavored concurrent object systems exist for, e.g., Java [3, 30], Eiffel [5, 26], and C++ [25]. Concurrent objects are reminiscent of Actors [1] and Erlang processes [2]: objects are inherently concurrent, conceptually each object has a dedicated processor, and there is at most one activity in an object at any time. Thus, concurrent objects encapsulate not only their state and methods, but also a single (active) thread of control. In the concurrent object model, *asynchronous method calls* may be used to better combine object-orientation with distributed programming by reducing the temporal coupling between the caller and callee of a method, compared

* Partly funded by the EU project FP7-610582 ENVISAGE

to the tightly synchronized (remote) method invocation model (of, e.g., Java RMI [27]). Intuitively, asynchronous method calls spawn activities in objects without blocking execution in the caller. Return values from asynchronous calls are managed by *futures* [14,23,32]. Asynchronous method calls and futures have been integrated with, e.g., Java [11,19] and Scala [13] and offer a large degree of potential concurrency for deployment on multi-core or distributed architectures.

ABS is a modeling language targeting distributed systems [17]; the language combines concurrent objects and asynchronous method calls with *cooperative scheduling* of method invocations. In ABS the basic unit of computation is the *concurrent object group* (cog): a cog provides to a group of objects a shared processor. Method invocations on an object of a cog instantiate a new task that requires the cog's processor in order to execute. Cooperative scheduling allows tasks to suspend in a controlled way at explicit points in the code, so that other tasks of the object can execute. The **suspend** and **await** commands are used to explicitly release the processor: the difference between the two commands is that **await** has an associated boolean guard expressing under which condition the task should be re-activated by the scheduler. Asynchronous method invocations are used among objects belonging to different cogs; at each asynchronous method invocation a *future* is instantiated to store the return value. Futures are first class citizens in ABS and are accessed via a **get** command; **get** is blocking because a task, executing **get** on a future of a method invocation which has not yet completed, blocks and keeps the processor until the future is written. To avoid keeping the processor, one can use an **await** `f?` to ensure that future `f` contains a value.

ABS has a formal, executable semantics; ABS models can be run on a variety of backends and can be verified using the KeY proof checker [4]. In particular, asynchronous method calls and cooperative scheduling allow the verification of distributed and concurrent programs by means of sequential reasoning [8]. In ABS this is reflected in a proof system for local reasoning about objects where the class invariant must hold at all scheduling points [9]. Although ABS targets distributed systems, a notable abstraction of the language design is that faults are currently not considered part of the behavior to be modeled. On the other hand, dealing with faults is an essential and notoriously difficult part of developing a distributed system; this difficulty is exacerbated by the lack of clear structuring concepts [7]. A well-designed model is essential to understand potential faults and reason about robustness, especially in distributed settings. Thus, it is interesting to extend a modeling language such as ABS in order to model faults and how these can be resolved during the system design.

It is common in the literature to distinguish errors due to the software design (sometimes called *faults*) from random errors due to hardware (sometimes called *failures*). For software deployed on a single machine, such hardware failures entail a crash of the program. A characteristic of distributed systems is that failures may be *partial* [31]; i.e., the failure may cause a node to crash or a link to be broken while the rest of the system continues to operate. In our setting, a strict separation between faults and failures may seem contrived, and

we will refer to unintended behavior caused by both the software and hardware as faults. A fault is *masked* if the fault is not detected by the client of the service in which the fault occurs. In hierarchical fault models, faults can propagate along the path of service requests; i.e., a fault at the server level can result in a (possibly different) fault at the client level. In a synchronous communication model, a client object can only send one method call at the time whereas in an asynchronous communication model, the client may spawn several calls. Thus, it need not be clear for a client object which of the spawned calls resulted in a specific fault in the asynchronous case. However, asynchronous method calls in ABS allow results to be *shared* before they are returned: futures are first-class citizens of ABS and may be passed around. First-class futures give rise to very flexible patterns of synchronization, but they further obfuscate the path of service requests and thus of fault propagation.

This paper discusses an extension of the semantics of the ABS modeling language to incorporate a robust fault model that is both amenable to formal analysis and familiar to the working programmer. The paper considers how faults can be introduced into ABS in a way which is faithful to its syntax, semantics, and proof system, and discusses the appropriate introduction of faults along three dimensions: fault representation (Section 2), fault behavior (Section 3), and fault propagation (Section 4).

2 How Are Faults Represented?

Exceptions are the language entities corresponding to faults in an ABS program's execution. ABS includes two kinds of entities which in principle can be used to represent faults: *objects* and *datatypes* (datatypes [16] are part of the functional layer of ABS, and abstract simple, common structures like lists and sets).

Exceptions as Objects. Representing exceptions as objects allows for a very flexible management of faults. Indeed, in this setting exceptions would have both a mutable state and a behavior. Also, one could define new kinds of exceptions using the interface hierarchy. Finally, exceptions would have identities allowing to distinguish different instances of the same fault. However, most of these features are not needed for faults: faults are generated and consumed, but they are static and with no behavior. Representing them as objects would allow a programming style not matching the intuition and difficult to understand. Furthermore, in ABS static verification is a main concern, and semantic clarity is more needed than in other languages. For this reason we think that in the setting of ABS exceptions should not be objects.

Exceptions as Datatypes. Datatypes fit more with the intuition of exceptions as described before: they are simple values with no identity nor behavior. However, in ABS datatypes are closed, meaning that once a datatype has been declared, it is not possible to extend it with new constructors. This is a potential problem in using them to represent exceptions. Indeed, we would like the datatype for

exceptions to include system-defined exceptions such as Division by Zero or Array out of Bound, and to be extended to accommodate user-defined exceptions. Also, for modularity reasons, programmers of an ABS module should be able to declare their own exceptions, thus exception declaration cannot be centralized. User-defined exceptions are not only handy for the programmer, but may also help the definition of invariants by tracking the occurrence of specific conditions. We discuss below a few possible design choices related to the definition of user-defined exceptions.

Allow open datatypes in ABS. In this setting exception would be an open datatype [24], and other ABS datatypes could be open as well. The declaration of system-defined exceptions can be done as:

```
open data Exception = NullPointerException
                    | RemoteAccessException
```

where the keyword **open** specifies that the datatype is open (in principle open and closed datatypes may coexist). Then one can add user-defined exceptions as:

```
open data Exception = ... | MyUserDefinedException
```

However, this is a major modification of datatypes, a key component of ABS, and introducing this additional complexity only to accommodate exceptions may not be a good choice. In fact, handling open datatypes is in contrast with the fact that ABS type system is nominal. One would need to resort to a structural type system (similar to, e.g., OCaml's variants [29]) to ensure that a pattern matching is complete, which is far less natural.

Allow any datatype to be an exception. In this setting any value of any datatype may be used as an exception (the fact of declaring which datatypes are actually used as exceptions does not change too much the setting). User-defined datatypes can be added by simply defining new datatypes. When the programmer wants to catch an exception, he has to specify which types of exceptions he can catch, and do a pattern matching both on the type and on its constructor to understand which particular fault happened. This produces a syntax like:

```
try { ... }
catch(List e) {
  case(e) {
    | Empty => ...
    | Cons(v,e2) => ...
} }
catch(NullPointerException e) { ... }
catch(_ e) { ... /* capture all exceptions */ }
```

where a special syntax _ is needed to catch exceptions of any type, since there is no hierarchy for datatypes in ABS. Note that in case the exception has type `List` a **case** is done to analyze its structure. A difficulty in applying this approach to ABS is due to the fact that in ABS values do not carry their type at runtime, but adding such an information seems not to have relevant drawbacks.

Exceptions as a new kind of value. In this setting exceptions are a separate kind of value, at the same level of objects and datatypes. The type Exception is open. New exceptions (both system- and user-defined) can be declared as follows:

```
exception NullPointerException
exception RemoteAccessException
...
exception MyUserDefinedException(Int, String)
```

Pattern matching can be used to distinguish different exceptions:

```
try { ... }
catch(e) {
  NullPointerException => ...
  MyUserDefinedException(n,s) => ...
  e2 => ...
  ...
}
```

Structural typing can be used if one wants to check that all exceptions possibly raised are caught, as, e.g., in Java.

Discussion. The simplest approach is to model exceptions as a closed datatype. However, if open exceptions are desired to increase the expressive power, the last solution is the one with minimal impact on the existing ABS language. Allowing any datatype as an exception also seems a viable option, but with a more substantial impact on the existing structure of ABS.

3 Which is the Behavior of Faults?

Faults interrupt the normal control flow of the program. A first issue concerning faults is how they are generated. Concerning fault management, it is a common agreement that faults are manipulated with a **try**/**catch** structure, and we do not see any reason to change this approach in our design for ABS. However, after this choice has been taken, the design space is still vast and many questions still need to be investigated.

Fault Generation. In programming languages, faults can be generated either by an explicit command such as **throw** `f` where `f` is the raised fault, or by a normal command. For instance, when evaluating the expression `x/y` a Division by Zero exception may be raised if `y` is 0. In this second case, which exception is raised is not explicit, but defined by the semantics of the command. After having been raised, the two kinds of exceptions are indistinguishable. A third kind of exception may be considered in ABS. Indeed, ABS is currently evolving towards having an explicit distribution, and in this setting localities or links may break. The only remote interaction in ABS is via asynchronous method invocation, and the corresponding **await**/**get** on the created future. In principle, network problems could be notified either during invocation, or during the **await**/**get**.

However, invocation is asynchronous, and will not check for instance whether the callee will receive and/or process the invocation message. For the same reason, it is not reasonable that it checks for network problems. Clearly, the **get** should raise the fault, since no return value is available.

The behavior of **await** depends on its intended semantics. If executing the statement **await** `f?` means that the process whose result will be stored in `f` has successfully finished, then the **await** needs to synchronize with the remote computation and should raise a pre-defined fault upon timeout and network errors. In this setting thus network faults are raised by both **await** and **get**. On the other hand, if executing **await** `f?` gives only the guarantee that a subsequent `f.`**get** will not block, then all faults, including network- and timeout-related faults, can be raised by **get** exclusively.

Fault Management. As discussed in the beginning of the section, we use the common **try**/**catch** structure to manage faults. This structure sometimes also features an additional block **finally**. The **finally** block specifies some code that must be executed both if no exception is raised and if it is. A common use of the **finally** block is to release resources which need to be freed whatever the result of the computation is.

```
try { ... }
catch(MyFirstException e) { ... }
catch(MySecondeException e) {... }
finally { P }
```

For instance, `P` may close a file used inside the **try** block. The **finally** block is very convenient for programming, but may not be needed in the core language. Indeed, in many cases it can be encoded. The encoding instantiated on the example above is as follows:

```
try {
  try { ... }
  catch(MyFirstException e) { ... }
  catch(MySecondeException e) { ... }
} catch(_ e) {
  P
  throw e;
}
P
```

Essentially, one has to catch all the exceptions, do `P` and rethrow the same exception. `P` also needs to be replicated at the end, so to be executed if no exception is raised. Note that this encoding relies on always having exactly one **return** statement per method, at its end (this is the recommended style of programming in ABS), and on the ability to catch all exceptions and to be able to rethrow them identically. Actually, in principle, one can also consider some uncatchable faults, but this seems not particularly relevant in practice.

For resource management, an alternative to the **finally** block is the autorelease mechanism of Java 7 [28], which automatically releases its resource at the end of

the block. Encoding such a mechanism in ABS could be done using an approach similar to the one above for the **finally** block.

Fault Effects. We have discussed how to catch faults. However, it may happen that a fault is not caught inside the method raising it. Then, as already said, the fault should interrupt the normal flow of computation, i.e. killing the running process. However, one may decide to kill a larger entity. Suitable candidates in ABS are the object where the fault has been raised, or its cog. Now, remember that in ABS there is a strong emphasis on correctness proofs based on invariants, and that whenever a process releases the lock of an object the class invariant must hold. An uncaught fault releases the lock by killing the running process. This means that whenever an uncaught fault may be raised, the invariant must hold. Since faults may be raised by many constructs, including expressions and **get**, ensuring this may be particularly difficult, and may require in practice to manage all the faults inside the method raising them. However, this is undesirable since a method may not have enough information to correctly manage a given fault. One can try to define a weaker invariant, but this may be difficult. A solution is to decide that a fault may not only kill the process, but also the object whose invariant may be no more valid. An even more drastic solution is to kill the whole cog. This may be meaningful if invariants involving different objects (of the same cog) are considered. However, this kind of invariant is currently not considered in ABS, thus the introduction of mechanisms for killing a whole cog seems premature.

Effect Declaration. In classic programming languages, the only effect of an uncaught fault is to kill the running process. However, we just discussed that also killing the whole object (or cog) is a possible effect. One may want to have different effects for different faults. More in general, different faults may have different properties. Another possible property may describe whether a fault can be caught or not. Whatever the set of possible properties is, an important issue is where those properties are associated with the raised fault. One can have a keyword **deadly** specifying that a given exception will kill the whole object if uncaught, while the behavior of just killing the process can be considered the default behavior. We can see three possibilities here. Properties may be specified:

when an exception is declared: for instance, one may write

```
deadly exception NullPointerException
```

A main drawback of this approach is that the same exception will behave the same everywhere. Intuitively, an exception may be deadly for an object where the invariant cannot be restored, and not for another one where the fault has no impact on the invariant. Note also that if any datatype can be an exception, then one has to specify properties for each datatype, e.g. **deadly** `Int`. Actually, this second drawback is mitigated by choosing suitable default values for properties.

when an exception is raised: for instance, one may write

```
throw deadly NullPointerException
```

or also shorten it into **die** `NullPointerException`. Clearly, this approach is only reasonable for exceptions raised by an explicit throw (unless one wants to write something like `x=y/0` **deadly**). The approach is also less compositional, thus less suitable for static analysis. In fact, to understand the behavior of an exception it is not enough to look at declarations. For instance, the same exception may be either deadly or not for the same method, depending on how it has been raised. Note also that this approach would break the encoding of **finally** above, since there is no way to rethrow an exception with the exact same properties.

in the signature of the method raising the exception: for instance, one may write

```
Int calc(Int x) deadly: NullPointerException {...}
```

Clearly, this approach is viable only for properties relevant when the exception exits the method, such as deadly. It would not work for instance to specify whether an exception can be caught or not. Notice that this approach integrates well with the declaration of which faults a method may raise, useful to statically verify that all exceptions are caught. In fact, one could write

```
Int calc(Int x) throws: DivisionByZero,
                  deadly NullPointerException {...}
```

More in general, this approach is suitable for static analysis, since a method declaration also provides the information on the behavior of exceptions raised by the method itself. The same information is useful also for the programmer, in particular when using methods he did not write himself.

Discussion. We think that in the context of ABS, a fault may have two different effects: either killing the process or the whole object, depending on whether the object invariant holds or not. Whether a fault should kill the whole object or not should be declared at the level of method signature to enhance compositionality. Note that in the most used object-oriented languages, objects are never killed as a result of an exception: indeed such a feature is relevant in ABS because of its emphasis on analysis based on invariants, and no widespread object-oriented language has been developed according to this philosophy. A possible alternative to kill the object would be to roll back state changes. A transparent rollback [10] in our setting could lead to the last release point, where one is sure the invariant holds. However such an approach, discussed in [12], is not always satisfying. Indeed, rolling back only locally may easily lead to inconsistencies between different local states (what corresponds to break invariants concerning multiple objects). On the other hand, global rollback as in [21] results in an overly complex semantics. Furthermore, if local rollback is needed in particular cases, it can usually be encoded. Similarly, the **finally** construct is not strictly needed, since with the choices we advocate it can be encoded.

4 How Do Faults Propagate?

We have discussed in the previous section the effect of a fault on the process or object where it is raised. However, in case of fault, in particular of uncaught fault, it is reasonable to propagate the exception also to other processes/objects related to it. In particular, possible targets for propagation are processes interested in the result of the computation, processes that have been invoked by the failed one, processes in the same object/cog of the failed one, processes trying to access an object after it died.

Propagation through the Return Future. In a language with synchronous method invocation the only process that can directly access the result of the computation is the caller. However, in languages with asynchronous method invocation any process receiving the future can directly access the result of the computation. The caller may be or may not be one of them, and indeed may even terminate before the result of the computation becomes ready. Thus we discuss here notification of faults to the processes synchronizing with the future. We have two possibilities: processes may synchronize with the future either with a **get** or with an **await** statement. The case of **get** is clear: those processes are interested in the result of the computation, in case of fault no correct result is available and those processes need to be notified so that they can decide how to proceed. The natural way of being notified is that the same exception is raised by **get**. A process doing an **await** is just interested in waiting for the computation to terminate, but not in knowing its result. Thus we claim that if the computation terminated, either with a normal value or with an exception, the **await** should not block and the exception, if any, should not be raised. The exception would be raised only if later on a **get** on the future is performed. This approach requires to put the fault notification inside the future, and has been explored in the context of ABS in [18]. Indeed, this is also the approach of Java future library (asynchronous computation with futures has been standardized in a Java library since Java SE 5 [11]). In contrast to ABS, Java's API does not distinguish between waiting for a future to become available, and retrieving the results. In fact, no primitive like **await** is available in Java. In addition, Java's futures do not faithfully propagate exceptions: the **get** method on a faulty future always raises the same exception `ExecutionException`.

An additional problem is related to concurrency. Indeed, in ABS, one may have multiple concurrent **get** and/or **await** statements on the same future containing an exception. Let us consider the case of multiple **get** statements. In this case, one has to decide whether they all raise the fault contained in the future or just one of them does. This second solution is more troublesome since to this end, the first process accessing the future would receive the exception and remove the fault from the future. The only possibility is to replace it with some default value, and this requires locking the future. However, this in turn changes the behavior of futures in a relevant way: Futures are understood as logical variables that change at most once, and this would no longer be true. Additionally, this creates a weak synchronization point between two processes

accessing the same future. Indeed, if a process knows that a future originally contains an exception, by accessing it he will know if another process accessed it before. These weak synchronization points between processes that would be independent otherwise make the concurrency model and thus the analysis more difficult. Note that concurrent **await** statements are not a problem, since they do not locally raise the fault, but just check whether the future is empty or not.

Propagation through Method Invocations. It may be the case that the failed computation has invoked methods in other objects, whose execution is no more necessary after the failure. Indeed, it may even be undesired. For instance, if you are planning a trip and the booking of the airplane fails, you do not want to complete the booking of the hotel. Thus a mechanism to cancel a computation originated by a past method call may be useful. Actually, cancel may have different meanings according to the state of the invocation. If the invocation has not started yet, one can simply remove the invocation message itself. If the invoked process is running, one may raise the exception. If the execution already completed, one may do nothing or execute some code to compensate the terminated execution. This second option has been explored in [18]. The most interesting case is the one where the invoked process is running. Indeed, in this case the fault may be raised in any point of the execution, thus dealing with it using a try-catch would require to have the whole method code, including the return command, inside the try block. A better approach is to define specific points in the code where the running process checks for exceptions from its invoker, and specifying there the code to be executed in this case. A more modular way is to separate the two issues. One may have a statement `check` to specify when to check for faults, and a statement **setHandler** `H` establishing that `H` is the handler to be used to deal with faults from the invoker from now on. `H` can be a simple piece of code, or a function associating pieces of code to exceptions. Pieces of code may have a return statement, to communicate the result of the fault management to the invoker. If the execution of the handler terminates successfully, the execution of the method code restarts. One may also decide that the last handler has to be used to compensate the execution if the cancellation occurs after the termination of the invoked process.

We have described the effect of propagation to invoked processes. However, one has to understand which invoked processes to consider. The simplest possibility is to let the programmer decide. We call this approach *programmed propagation*. This can be done through a statement `f1 = f.`**cancel**`(e)` where `f` is the future corresponding to the invocation to be canceled, `e` the exception to be raised and `f1` the future storing the result of exception management. Note that the future `f` is the right entity to individuate the invocation, since each invocation corresponds to a different future. Note also that with programmed cancel one may cancel twice the same invocation, and that cancel can be executed by any process on any future he knows of. Future `f1` may contain different values according to the outcome of the cancel. If the exception sent by the cancel is correctly managed, the handler returns a specific value to fill that future (potentially different from the value returned as a result of the method, which is in

future `f`). In all the other cases a system-defined exception is put inside future `f1` (one cannot put there a normal value, since this should depend on the type of the future):

- an exception `notStarted` if the **cancel** arrives before the invocation started (while the future `f` contains an exception `canceled`);
- an exception `terminated` if the **cancel** arrives after method termination, and compensations are not used (future `f` keeps its value);
- an exception `noCompensation` if the **cancel** arrives after method termination, compensations are allowed, but no compensation is specified for the target method (future `f` keeps its value);
- an exception `CancelNotManaged` if the exception arrives when the method is running, but it is not managed since there is no handler for it (while the future `f` stores the exception `e`).

In case of multiple cancellations, cancellations behave as above according to the state of the method when they are processed. Note also that the future `f` is not changed if it already contains the result of the method invocation.

An alternative approach is to have an *automatic propagation* of exceptions to invoked processes. First, one should decide whether to propagate only uncaught faults, or also managed faults. This last solution is not desirable in general, since most managed faults should not affect other processes, and can be dealt with by programmed propagation in case of need. Propagation of uncaught faults, if desired, should be necessarily done automatically. Now, the problem is to understand to which method invocations the fault needs to propagate. An upper bound is given by the futures known by the dying process. One may also consider that futures on which a **get** has already been performed are no more relevant. However, there is no fast and easy answer to this question. We think that a reasonable solution is to choose the futures which have been created by the current method execution and on which no **get** has been performed yet by the same method. One may also want to check whether the reference to the future has been passed to another method, and whether this method has performed a **get** on the future, but this would make the implementation and the analysis much more tricky. Similarly, one may want to consider also futures received as parameters, but again this needs to propagate the information on whether a future has been accessed or not from one method to the other. Note that in case of automatic propagation, no information on the result of the cancellation is needed, since the caller already terminated.

One may want to ensure that children can manage all the faults from their parent. To this end, each child should declare the exceptions that he can manage (at any point, since it may be the case that some exceptions can be managed only at some `check` due to handler modifications). Then, one can check that these include all the exceptions the parent may send to him. For automatic propagation, these coincide with the exceptions the parent may raise. For programmed propagation, these are the arguments of the various **cancel** of the corresponding future in the parent or in methods to which the future is passed.

Propagation to Other Processes in the same Object/Cog. We already discussed the fact that it is important for processes to restore the invariant of the object or cog before releasing the lock, and in particular before terminating because of an uncaught exception. In case the invariant cannot be restored, we proposed as a solution the possibility of killing the whole object/cog. An alternative solution is to terminate only the process P that first raised the fault, and notifying the other processes about the uncaught fault, since they may be able to manage it. Note that when process P is terminated, there is no other running process in the same cog. Thus the other processes will get the fault notification when they will be scheduled again. This means that they may get a fault, either when they start, or when they resume execution at an **await** or **suspend** statement. The fault may then be managed, or propagated as discussed above. In particular, if not managed, will be propagated to the next process to be scheduled. In this setting objects never die, but method calls may receive an exception as soon as they start. If we raise the same exception that was raised by P, then it may be difficult to track which exceptions may be raised inside any method. A simpler solution is to have a dedicated exception, e.g., InvariantNotRestored. What said above for objects holds similarly for cogs, concerning cog invariants. However, as already said, they are not a main concern in ABS at the moment.

Propagation through Dead Objects/Cogs. Some of the approaches we discussed involve the killing of an object or cog. We have not yet discussed what happens when a dead object is accessed (through method invocation). An exception should be raised. We can follow either the approach discussed above for network errors, or the one for normal faults. In practice the only difference is whether also the **await** will raise such a fault or not. We do not see any particular advantages or disadvantages for the two approaches: which is the best solution depends on which one better fits the programmer intuition, which may be different from one programmer to the other. In both the cases, using a standard exception such as DeadObject, instead of propagating the exception that caused the death of the object, simplifies the management. Also, it allows the caller to know whether the object is dead because of its invocation or it was already dead before.

Discussion. Among the propagation strategies above, propagation through the return future is nowadays standard, since it is used, e.g., in Java and C#. The possibility of canceling a running process via a future is also available in Java and C#, but the possibility of doing it automatically and/or of defining handlers and compensations for managing cancel requests while the process is running are not considered. Indeed, these strategies are quite complex, and it is not yet clear how useful they are in real programming. Also propagation of the fault to other processes in the same object is not considered in mainstream languages as far as we know, but we think this is a viable strategy in ABS. In fact, when a process is not able to restore the object invariant, there are two possibilities: either destroying the whole object or leave to another method call the task of restoring it. This second strategy seems also less extreme.

5 Conclusion

We have discussed the design space for fault models to be included in the ABS modeling language. As future work, we will extend the formal semantics of ABS with appropriate datatypes for the representation of faults, primitives to raise and catch faults, and mechanisms to distribute faults to other objects and cogs.

We complete the paper with a review of related fault models (the comparison with Java has been already discussed throughout the paper).

Functional programming languages, like OCaml or Haskell, also include primitives for faults modeled as exceptions. Both languages allow user-defined exceptions, but they implement them in different manners. OCaml uses a special type (called *exn*) to type exceptions, and new exceptions can be declared using the syntax **exception** `e` **of** `data`. In [24], the introduction of open datatypes in Haskell to encode exceptions is discussed. However, the current implementation of GHC uses *typeclass* [6], which allow one to register new datatypes as being exceptions.

Message-Passing Interface (MPI) is a cross-language standard, used, e.g., for C and Fortran, to program distributed applications. MPI expresses communication via so-called MPI functions, the basic ones being SEND and a blocking RECV. The SEND can work with three modalities: (*buffered send*) buffering the data to be sent, thus returning immediately as we assumed in this paper; (*synchronous send*) waiting for a corresponding RECV to be posted by the destination before terminating; or (*ready send*) failing in the case a corresponding RECV has not yet been posted by the destination. Dealing with the network, MPI functions represent communication at a lower level than we do: in MPI also the process of data delivering is taken into account. SEND (in all its modalities) and RECV have asynchronous variants, called ISEND and IRECV (where the "I" stands for "initiation"), which indicate a buffer where to fetch/put data and return immediately. For each such asynchronous send/receive, functionalities similar to some of the ones considered in this paper can be used: WAIT makes it possible to wait for the completion of data sending/receiving (in addition in MPI there is also a function TEST which returns immediately the status of the data sending/receiving without waiting); failures (e.g., in the communication while sending/receiving data) are detected by calling WAIT or TEST; and it is possible to cancel a send/receive by a call to CANCEL. The semantics of the latter, however, is the removal of the send/receive, supposing that it has not completed yet, as if it never occurred (a matching receive/send would perceive, as well, the canceled send/receive as if it never occurred). By combining the mechanisms above it is possible to obtain the waiting and canceling mechanisms considered in this paper. For example an asynchronous method invocation can be modeled by executing both ISEND and IRECV, and cancellation by executing CANCEL both on the send and the receive in the case the data is still under transmission or just on the receive in the case the send has completed. In the latter case, if the invoked method performs a ready send at the end of method execution, it will be notified of the matching IRECV having been canceled.

In web applications the HTTP protocol is used to realize service invocations by means of request/response pairs over a TCP/IP connection, as happens in

the popular approaches of Java and Javascript, i.e. Asynchronous Javascript and XML (AJAX) invocations. In Java a method is used to initiate the HTTP request/response, which differently from the approach considered in this paper, may yield an error in the case the connection with the HTTP server cannot be established (timeout based). Then the client goes through a two phase process, where he first sends data over an output-stream and then, similarly, receives data. At the server side a symmetric process is followed. Java methods for reading pieces of response data are blocking as for the waiting function used in MPI and considered in this paper. Similarly, failures are notified via exceptions when reading response data (or while sending request data). Finally concerning cancellation, the HTTP request/response can be aborted as a whole by the client and this causes the server to detect the failure (an exception is raised) when it is in the phase of inserting data in the response, i.e. when returning data (or while reading request data). In Javascript (AJAX) request data are preliminarily put into memory (as in MPI) and then the request/response is initiated (again this can fail if connection cannot be established). Such an initiation function also installs a user-defined function which is expected to manage the data received once the response is completed (including also managing the case of failure). This mechanism is an alternative to the waiting function used in MPI and considered in this paper. Concerning cancellation, it is possible, as in Java, to abort the HTTP request/response (with the same effect at server side).

References

1. G. Agha and C. Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In *Research Directions in Object-Oriented Programming*, pages 49–74. MIT Press, 1987.
2. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
3. L. Baduel et al. *Grid Computing: Software Environments and Tools*, chapter Programming, Composing, Deploying, for the Grid. Springer, 2006.
4. B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of Object-oriented Software: The KeY Approach*. Springer, 2007.
5. D. Caromel. Service, Asynchrony, and Wait-By-Necessity. *Journal of Object Oriented Programming*, pages 12–22, 1989.
6. K. Chen, P. Hudak, and M. Odersky. Parametric type classes. In *Proc. of LFP'92*, pages 170–181. ACM, 1992.
7. F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.
8. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *ESOP*, volume 4421 of *LNCS*, pages 316–330. Springer, 2007.
9. C. C. Din, J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. *Journal of Logic and Algebraic Programming*, 81(3):227–256, 2012.
10. E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

11. B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.
12. G. Göri, E. B. Johnsen, R. Schlatte, and V. Stolz. Erlang-style error recovery for concurrent objects with cooperative scheduling. In *ISoLA*, LNCS. Springer, 2014. To appear.
13. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009.
14. R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Prog. Lang. Syst.*, 7(4):501–538, 1985.
15. International Telecommunication Union. Open Distributed Processing — Reference Model parts 1–4. Technical report, ISO/IEC, Geneva, July 1995.
16. B. Jay. Algebraic data types. In *Pattern Calculus*, pages 149–160. Springer, 2009.
17. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Proc. of FMCO 2010*, volume 6957 of *LNCS*, pages 142–164. Springer, 2011.
18. E. B. Johnsen, I. Lanese, and G. Zavattaro. Fault in the future. In *COORDINATION*, volume 6721 of *LNCS*, pages 1–15. Springer, 2011.
19. *JSR166: Concurrency utilities*. `http://java.sun.com/j2se/1.5.0/docs/guide/concurrency`.
20. J. Kramer. Is abstraction the key to computing? *Communications of the ACM*, 50(4):36–42, 2007.
21. I. Lanese, C. A. Mezzina, A. Schmitt, and J.-B. Stefani. Controlling reversibility in higher-order pi. In *CONCUR*, volume 6901 of *LNCS*, pages 297–311. Springer, 2011.
22. R. G. Lavender and D. C. Schmidt. Active object: an object behavioral pattern for concurrent programming. In *Pattern languages of program design 2*, pages 483–499. Addison-Wesley Longman Publishing Co., Inc., 1996.
23. B. H. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI*, pages 260–267. ACM Press, 1988.
24. A. Löh and R. Hinze. Open data types and open functions. In *Proc. of PPDP'06*, pages 133–144. ACM, 2006.
25. B. Morris. CActive and Friends. Symbian Developer Network, November 2007. `http://developer.symbian.com/main/downloads/papers/CActiveAndFriends/CActiveAndFriends.pdf`.
26. P. Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming*. PhD thesis, Department of Computer Science, ETH Zurich, 2007.
27. E. Pitt and K. McNiff. *Java.Rmi: The Remote Method Invocation Guide*. Addison-Wesley Longman Publishing Co., Inc., 2001.
28. J. Ponge. Better resource management with Java SE 7: Beyond syntactic sugar, May 2011. `http://www.oracle.com/technetwork/articles/java/trywithresources-401775.html`.
29. D. Rémy. Type checking records and variants in a natural extension of ml. In *Proc. of POPL'89*, pages 77–88. ACM, 1989.
30. J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *ECOOP*, volume 6183 of *LNCS*, pages 275–299. Springer, 2010.
31. J. Waldo, G. Wyant, A. Wollrath, and S. C. Kendall. A note on distributed computing. In *MOS'96)*, volume 1222 of *LNCS*, pages 49–64. Springer, 1997.
32. A. Yonezawa. *ABCL: An Object-Oriented Concurrent System*. MIT Press, 1990.

A.3 Erlang-style Error Recovery for Concurrent Objects with Cooperative Scheduling

Erlang-style Error Recovery for Concurrent Objects with Cooperative Scheduling *

Georg Göri[1], Einar Broch Johnsen[2], Rudolf Schlatte[2], and Volker Stolz[2]

[1] University of Technology, Graz, Austria
goeri@student.tugraz.at

[2] University of Oslo, Norway
{einarj,rudi,stolz}@ifi.uio.no

Abstract. Re-establishing a safe program state after an error occurred is a known problem. Manually written error-recovery code is both more difficult to test and less often executed than the main code paths, hence errors are prevalent in these parts of a program. This paper proposes a failure model for concurrent objects with cooperative scheduling that automatically re-establishes object invariants after program failures, thereby eliminating the need to manually write this problematic code. The proposed model relies on a number of features of actor-based object-oriented languages, such as asynchronous method calls, co-operative scheduling with explicit synchronization points, and communication via future variables. We show that this approach can be used to implement Erlang-style process linking, and implement a supervision tree as a proof-of-concept.

1 Introduction

Crashes and errors in real-world systems are not always due to faulty programming. Especially but not only in distributed systems, error conditions can arise that are not a consequence of the logic of the running program. Robust systems must be able to deal with and mitigate such unexpected conditions. At the same time, error recovery code is notoriously hard to test.

An influential approach to more robust systems is "Crash-Only Software" [4], i.e., letting system components fail and restarting them. Erlang [2,19] is a widely-used functional language which successfully adopts these ideas. However, inherent in such subsystem restarts is the accompanying loss of state. This is much less a problem with programs written in a functional style than with programs written using object-oriented techniques, where the objects themselves hold state. This paper describes an approach to crash-only software which can keep objects alive without explicit code to restore object invariants.

The approach of this paper is based on concurrent objects which communicate by means of *asynchronous method calls*; the caller allocates a *future* as a container for the forthcoming result of the method call, and keeps executing until the result of the call is needed. Since execution can get stuck waiting for a reply,

* Partially funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu).

we allow process execution to suspend by introducing processor release points related to the polling of futures. Scheduling is *cooperative* via release-points in the code, awaiting either a condition on the state of the object, or the availability of the result from a method call. To concretize the approach, we use some features from the abstract behavior specification language ABS [11], a statically typed object-oriented modeling language targeting distributed systems. ABS has a formal semantics implemented in the rewriting logic tool Maude [7], which can be used to explore the runtime behavior of specifications.

This paper introduces linguistic means to both abort a single computation without corrupting object state and to terminate an object with all its pending processes. We provide a formal semantics for how those faults propagate through asynchronous communication. Callers may decide to not care about faults and fail themselves when trying to access the result of a call whose computation aborted, or use a safe means of access that allows them to explicitly distinguish a fault from a normal result and react accordingly. We show the usefulness of the new language primitives by showing how they allow us to implement process linking and supervision hierarchies, the standard recovery features of Erlang.

The rest of the paper is organized as follows. Section 2 describes the ABS language, Section 3 the novel failure model. Section 4 presents an operational semantics of a subset of the language, and illustrates the new functionality by modeling Erlang's well-known supervision architecture, and Section 5 discusses related and future work.

2 Behavioral Modeling in ABS

ABS is an abstract, executable, object-oriented modeling language with a formal semantics [11], targeting distributed systems. ABS is based on concurrent objects [5, 13] communicating by means of asynchronous method calls. Objects in ABS support interleaved concurrency based on explicit scheduling points. This allows active and reactive behavior to be easily combined, by means of a cooperative scheduling of processes which stem from method calls. Asynchronous method calls and cooperative scheduling allow the verification of distributed and concurrent programs by means of sequential reasoning [8]. In ABS this is reflected in a proof system for local reasoning about objects where the class invariant must hold at all scheduling points [9].

ABS combines functional and imperative programming styles with a Java-like syntax. Objects execute in parallel and communicate through asynchronous method calls. However, the data manipulation inside methods is modeled using a simple functional language based on user-defined algebraic data types and functions. Thus, the modeler may abstract from the details of low-level imperative implementations of data structures while maintaining an overall object-oriented design close to the target system.

The Functional Layer. The functional layer of ABS consists of algebraic data types such as the empty type `Unit`, booleans `Bool`, and integers `Int`; parametric

$$
\begin{array}{rl}
T & ::= I \mid D \mid D\langle\overline{T}\rangle \\
A & ::= X \mid T \mid D\langle\overline{A}\rangle \\
Dd & ::= \textsf{data}\ D[\langle\overline{A}\rangle] = [\overline{Cons}]; \\
Cons & ::= Co[(\overline{A})] \\
F & ::= \textsf{def}\ A\ fn[\langle\overline{A}\rangle](\overline{A}\,\overline{x}) = e; \\
e & ::= x \mid v \mid Co[(\overline{e})] \mid fn(\overline{e}) \mid \textsf{case}\ e\ \{\overline{br}\} \\
v & ::= Co[(\overline{v})] \mid \textsf{null} \\
br & ::= p \Rightarrow e; \\
p & ::= \_ \mid x \mid v \mid Co[(\overline{p})]
\end{array}
\qquad
\begin{array}{rl}
P & ::= \overline{IF}\ \overline{CL}\ \{[\overline{T}\ \overline{x};]\ s\} \\
IF & ::= \textsf{interface}\ I\ \{\,[\overline{Sg}]\,\} \\
CL & ::= \textsf{class}\ C\,[(\overline{T}\ \overline{x})]\,[\textsf{implements}\ \overline{I}]\,\{\,[\overline{T}\ \overline{x};]\,\overline{M}\} \\
Sg & ::= T\ m\ ([\overline{T}\ \overline{x}]) \\
M & ::= Sg\ \{[\overline{T}\ \overline{x};]\ s\} \\
g & ::= b \mid x? \mid g \wedge g \\
s & ::= s; s \mid \textsf{skip} \mid \textsf{if}\ b\,\{\,s\,\}\,[\textsf{else}\,\{\,s\,\}] \mid \textsf{while}\ b\,\{\,s\,\} \\
 & \quad \mid \textsf{suspend} \mid \textsf{await}\ g \mid x = rhs \mid \textsf{return}\ e \\
rhs & ::= e \mid cm \mid \textsf{new}\ C\,(\overline{e}) \\
cm & ::= [e]!m(\overline{e}) \mid x.\textsf{get}
\end{array}
$$

Fig. 1. ABS syntax for the functional (left) and imperative (right) layers. The terms $\overline{e}$ and $\overline{x}$ denote possibly empty lists over the corresponding syntactic categories, and square brackets [ ] optional elements.

data types such as sets `Set<X>` and maps `Map<X>` (for a type parameter `X`); and functions over values of these data types, with support for pattern matching.

The syntax of the functional layer is given in Figure 1 (left). The ground types T are interfaces I, type names D, and instantiated parametric data types $D\langle\overline{T}\rangle$. Parametric data types A allow type names to be parameterized by type variables X. User-defined data types definitions Dd introduce a name D for a new data type, parameters $\overline{A}$, and a list of constructors $Cons$. User-defined function definitions F have a return type A, a name fn, possible type parameters, a list of typed input variables x, and an expression e. Expressions e are variables x, values v, constructor, functional, and case expressions. Values v are constructors applied to values, or `null`. Case expressions match an expression e to a list of case branches br on the form $p \Rightarrow e$ which associate a pattern p with an expression e. Branches are evaluated in the listed order, the (possibly nested) pattern p includes an underscore which works as a wild card during pattern matching; variables in `p` are bound during pattern matching and are in the scope of the branch expression e. ABS provides a library with standard data types such as booleans, integers, sets, and maps, and functions over these data types.

The functional layer of ABS can be illustrated by considering naive *polymorphic sets* defined using a type variable `X` and two constructors `EmptySet` and `Insert`:

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

Two functions `contains`, which checks whether an item `el` is an element in a set `set`, and `take`, which selects an element from a non-empty set `set`, can be defined by pattern matching over `set`:

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

def X take<X>(Set<X> set) = case set { Insert(e, _) => e; };
```

The Imperative Layer. The imperative layer of ABS addresses concurrency, communication, and synchronization at the level of objects, and defines interfaces, classes, and methods. In contrast to mainstream object-oriented languages, ABS does not have an explicit concept of threads. Instead a thread of execution is unified with an object as the unit of concurrency and distribution, which eliminates race conditions in the models. Objects are *active* in the sense that their `run` method, if defined, gets called upon creation.

The syntax of the imperative layer of ABS is given in Figure 1 (right). A program P lists interface definitions IF and class definitions CL, and has a main block $\{[\overline{T}\ \overline{x};]\ s\}$ where the variables x of types T are in the scope of the statement s. Interface and class definitions, as well as signatures Sg and method definitions M are as in Java. As usual, **this** is a read-only field of an object, referring to the identifier of the object; similarly, we let **destiny** be a read-only variable in the scope of a method activation, referring to the future for the return value from the method activation. Below we focus on explaining the asynchronous communication and suspension mechanisms of ABS.

Communication and synchronization are decoupled in ABS. Communication is based on asynchronous method calls, denoted by assignments `f=o!m(e)` where `f` is a future variable, `o` an object expression, and `e` are (data value or object) expressions. After calling `f=o!m(e)`, the caller may proceed with its execution *without blocking* on the method reply. Two operations on future variables control synchronization in ABS. First, the statement **await** `f?` *suspends the active process* unless a return value from the call associated with `f` has arrived, allowing other processes in the same object to execute. Second, the return value is retrieved by the expression `f.`**get**, which *blocks all execution in the object* until the return value is available. Inside an object, ABS also supports standard synchronous method calls `o.m(e)`.

Objects locally sequentialize execution, resembling a monitor with release points but without explicit signaling. An object can have at most one active process. This active process can be unconditionally suspended by the statement **suspend**, adding this process to the queue of the object, from which an enabled process is then selected for execution. The guards `g` in **await** `g` control suspension of the active process and consist of Boolean conditions b conjoined with return tests `f?` on future variables `f` and with time-bounded suspensions `duration(e1,e2)` which become enabled between a best-case `e1` and a worst-case `e2` amount of time. Just like functional expressions, guards `g` are side-effect free. Instead of suspending, the active process may *block* while waiting for a reply as discussed above, or it may block for some amount of time between a best-case `e1` and a worst-case `e2`, using the syntax `duration(e1,e2)` [3]. The remaining statements of ABS are standard; e.g., sequential composition $s_1; s_2$, assignment `x=rhs`, and **skip**, **if**, **while**, and **return** constructs. Right hand side expressions `rhs` include the creation of an object **new** `C(e)`, method calls, and future dereferencing `f.`**get**, in addition to the functional expressions `e`.

Example. To illustrate the imperative layers of ABS, let us consider an interface `Account`, with methods `deposit` and `withdraw`, which is implemented by a class

```
interface Account {
    Unit deposit (Int amount);
    Unit withdraw (Int amount);
}

class Account implements Account {
    List<Int> transactions = Nil; // log of transactions
    Int balance = 0; // current balance
    Unit deposit (Int amount) {
        transactions = Cons(amount, transactions);
        balance = balance + amount;
    }
    Unit withdraw (Int amount) {
        transactions = Cons(-amount, transactions);
        if (balance < amount) abort "Insufficient funds";
        balance = balance - amount;
    }
}
```

Fig. 2. Bank account with history in ABS

`BankAccount` (as shown in Figure 2). We see that expressions from the functional layer are used inside the method implementations; e.g., the constructor `Cons` is used in the right hand side of an assignment to extend the list of transactions, and infix functions `+` and `-` are similarly used to adjust the balance.

To approach the theme of the next section, the example does not resolve the case of negative balance on the account (ignoring the issue of a better design which checks the condition before updating the history). A call to `withdraw` will only succeed if the balance is sufficient; if the `balance` is less than `amount` it is unclear what would be meaningful behavior in order to restore a class invariant like `balance`$\geq$`0`, and the method activation will *abort*: the previous state of the object will be restored, and the future storing the implicit return value of `Unit` type will be filled with a value indicating that an error occurred.

3 Failure Models and Error handling

Apart from user-specified aborts, it is very common for programs to run into so-called *runtime errors*, i.e., abnormal termination in a case where the operational semantics does not prescribe how the system can proceed. Prominent representatives of this class of faults are division by zero, null pointer accesses in languages that allow pointer dereferences, and errors that are propagated from the runtime system in managed languages, like out of memory errors when no more objects can be allocated.

In the semantics of the ABS language, behavior in those situations is underspecified, even though those situations can be encountered by the backends when running the code generated from an ABS model. For example, in the Maude semantics, a *division by zero* does not allow further reduction of that process, which may go unnoticed in the overall system, or lead to a deadlock when other processes wait on the object. In the Java backend, the underlying Java runtime

will generate a Java exception through the primitive math operations, which will terminate the current (ABS) process, and lead to similar effects as in Maude.

3.1 Design considerations

Invariants and the system. On abrupt termination of a computation, we need to establish which reaction would be required. In a distributed, loosely coupled system, a local error should not affect the complete system. So clearly here the guiding point must be that we have to keep the effects *local*. In our actor-based setting, we can take the locality even further: Although a computation failed, we can limit the effects to the *current process*. The object may still be able to process pending and future requests (although the caller of the failing process needs to be notified). But what should be the basis for further executions within this object?

The underlying motivation for the explicit release points in the language are of course the class invariants that developers rely on when designing their programs. As such, each method call expects that its respective object invariant holds upon entry (and upon awakening). This is clearly not the case under abrupt termination, before which the fields of the object may have been arbitrarily manipulated—the next release point may not have been reached.

Error handling in an object system. The mechanism we propose, defines the behavior in case of an error:

- Propagate errors through futures. The caller receives an error when reading the future.
- Default to having no explicit error handling, in which case a *process* is terminated, yet the *object* stays alive.
- Revert any partial state modifications to the current object up to the last release point.

These concepts are introduced by extending futures to propagate a possible error in the callee to the caller, providing a method to detect and handle an error contained in a future, and to terminate the caller in the case an error in a future is accessed by the default mechanism.

Linguistic support for error handling. We consider the following linguistic support to enable the envisaged error handling:

- a notion of user-defined error types
- a generalization of futures to either return values or propagate errors
- a statement **abort** `e`, which raises an error `e` and terminates the process
- a statement `f.`**safeget**, which can receive errors and values from a future `f`
- a statement **die**, which terminates the current object and all its processes

The occurrence of an error is represented in the model by means of the statement **abort** `e`, where `e` is an user defined error. These errors are represented by a special data type (see [15] for an extensive discussion of the potential design decisions). Such an abort can either be explicit in the model or can occur implicit either in internals of the execution, to represent distribution, system (e.g. out of memory) or runtime (e.g. division through zero) errors.

The semantic interpretation is dependent on the kind of ABS process the evaluation occurs in:

Active Object processes, represent the object's implicit execution of its `run` method. If in that process an **abort** `e` statement is evaluated, all current asynchronous calls to this object will abort with the error `e` and the references to this object will become invalid. Further synchronous or asynchronous calls to this object are equivalent to an **abort** `DeadObject` on the caller side. This mechanism was chosen, as the object behavior (its `run` method) is seen as an integral part of its correctness, and like an invalid state also an invalid termination of this behavior leads to an inconsistent object and therefore the object cannot be further used.

Asynchronous Call processes evaluated a method call in the called object. An **abort** `e` statement will terminate the process and return the error e to the associated future. Moreover, the callee will perform a rollback (see below).

Main Process. The main process (similar to Java's `main`-method entry point) represents the begin of the execution, and an **abort** there will, by convention, lead to the runtime system being terminated (in principle, this could be handled uniformly like the normal case, but in practice we prefer termination).

An automatic rollback discards all changes to the object's values since the last scheduling point, which can be either an **await** or **suspend**. This guarantees that objects only evolve from one state at a scheduling point to another, and not leave in case of an error an object in a state, which could violate the object invariant.

Extending futures to contain either the computed value or a potential error raised either by an **abort** on the callee side (or from the runtime in a distributed setting), enables error propagation over invocations. Following this, also the semantics of the `Future.`**get** statement needs to be adjusted: a **get** will, in presence of an error `e` in the future, lead to an implicit **abort** `e` on the caller side.

The newly introduced `Future.`**safeget** stops this propagation and allows one to react on errors. **safeget** returns a value of the algebraic data type `Result<T>`, which is defined as `Result<T> = Value(T val)|Error(String s)`. In case the future contains an error `e`, the same is returned, otherwise the constructor `Value(T v)` wraps the result value `v`. Note that due to the lack of subtyping in the type system, currently the only way to communicate an error indication is through a value of type `String`, as we cannot define a common type for all possible (incl. user-defined) errors.

The **die** `e` *statement* allows in asynchronous calls to terminate the active object. Its semantic meaning is the same as an **abort** `e` in the execution context of an active object process or init block. In other words, all pending asynchronous calls and the active object's process are terminated. This statement allows to implement linking (see below), and can be used in distributed models to simulate a disconnect from an object.

Discussion. We come back to the banking example in Listing 2 to illustrate the point of rollbacks. The general contract is that the list of `transactions` should accurately reflect the current total in the account. As the body of `withdraw` needs to modify two fields, we clearly benefit from ABS's semantics of explicit release points which guarantees that only one process is executing within the object (e.g. in Java, we would be required to explicitly declare the method as `synchronized` to achieve the same effect).

Nonetheless, even though if only by construction of the example, an **abort** would leave the object in an undesired state, as after the modification of the list of transactions the balance is no longer in sync with the banking transaction history. If an **abort** would simply terminate execution of the current process, and start processing another pending call on the current state of the object, we would observe invalid results. But with the rollback before processing another call, this assumption can easily be re-established.

Note that the ABS methodology is only concerned with *object* invariants, and this mechanism does not give us *totality* in the sense that a method either completes successfully or not at all: a rollback will not undo changes in other objects that have (transitively) occurred as the result of method calls during execution of the current process, unlike e.g. in work on a higher-order π-calculus [16]. This means on the one hand that the developer still has to actively take into account the workings of error recovery when designing the system, but on the other hand allows us to implement this feature efficiently by only keeping track of fields in the current object that are actually touched.

Compared to traditional object-oriented programming, we note that this implicit error handling strategy frees the developers from restoring state explicitly in an exception handler. However, through the **safeget** mechanism, they still have this option open.

3.2 A practical application of error propagation: process linking

The previously presented primitives enable an implementation of Erlang-style linking between two objects in ABS. These links are part of the foundation for Erlang's well known and successful error handling [1]. Erlang's communication model is even more loosely coupled than ABS, in that it is based on asynchronous message passing. As such, there are no method calls or explicit returns, but rather the callee has to send back a response, which will be queued in the recipient until extracted from the mailbox. Thus, a failure in the recipient process will either go unnoticed if no response messages are used or otherwise lead to an expected

message not being sent/received, and in turn a corresponding potential blockage can occur in the initial sender.

Erlang's links enable mutual observation of processes. A process can link itself to another process. If one of the two processes terminates, the runtime environment sends an *EXIT* message to the other process, which contains an exit reason. Unless this exit reason is *normal* (termination because the process reached the end of the function), the linked process will terminate as well, and in consequence propagate its own *EXIT* message to its linked processes. With this *error propagation*, it is possible to let groups of processes up to the whole system terminate automatically and clean up components consisting of multiple processes.

To enable processes to observe exit messages or react on them, a process can be marked to be a system process with the *trap_exit* process flag. Such processes will not terminate when receiving an *EXIT* message, but can retrieve this message from their inbox.

Implementation in the concurrent object model. The implementation idea is to represent a link by two asynchronous calls, one to each of the objects. Each call will only terminate upon termination of the object, and thus enables the caller to take an action.

In Figure 3 a sample implementation is shown, which assumes that each class implements code similar to the `Linkable` class. A link can be established by creating a new object of class `Link`, where the link gets initialized with references to both objects (referred to as `s` and `f`), and then calling `setup` on this new link. The `setup` method will initiate the calls between the objects, by calling `waitOn` and then wait until both calls are processed, where finished calls can be seen by the counter `done`.

The `waitOn` method implemented in the `Linkable` class places the normally non-terminating asynchronous call in line 3 to the other `Linkable` it should link to. The non-termination is achieved by a simple **await** `false`, as can be seen in the `wait` method. After those calls are made, the `waitOn` method reports back to the `Link` that it succeeded, and will afterwards await the termination of the call in line 3. The only possibility for a call to `wait` to return is when the object dies. Should now this future ever contain a value it must be an error, where in line 7 we can now take an action in case that the other object terminated, which will be in the default case a subsequent termination of the local object, by executing **die** `e`.

Linking in a producer consumer environment can be used to bind both objects together, so that a termination of the producer or consumer leads to the termination of the other party as well. In Figure 4 we see a `Producer` and `Consumer`, modeled as ABS classes, where the `Producer` sends a new input to the `Consumer` via an asynchronous call. Both classes have to implement the `Linkable` interface and include the shown default implementation of `wait` and `waitOn`.

Setting up a Link between `Producer` and `Consumer` is performed by the first two lines in the `Producer`'s `run` method. We construct the `Link` object and ini-

```
class Link(Linkable f,Linkable s)
  implements Link{
  Int done=0;
  Unit setup(){
   f!waitOn(this,s);
   s!waitOn(this,f);
   await done==2;
  }

  Unit done(){
   done=done+1;
  }
}
```

```
class Linkable() implements Linkable{
  Unit waitOn(Link l,Linkable la){
    Fut<Unit> fut=la!wait();
    l!done();
    await fut?;
    case fut.safeget {
     Error(e) => die e;
    }
  }
  Unit wait(){
    await false;
  }
}
```

Fig. 3. Implementation of links in ABS

```
class Producer(Consumer c)
  implements Linkable{
 Unit run(){
  Link lConsumer= new Link(this,c);
  await lConsumer!setup();
  // produce
  c!consume(X);
 }
 // include wait and waitOn
}
```

```
class Consumer()
  implements Linkable{

 Unit consume(String x){
    // consume
 }

 // include wait and waitOn,
}
```

Fig. 4. Links between a Producer and a Consumer

tialize the link via the `setup` method. A more detailed view of asynchronous calls and their lifetime is presented in the sequence diagram in Figure 5, arrows represent an invocation and a possible return value, and boxes represent the duration of the call on the callee side. First, the link is setup, two inputs are produced, and after that the `Consumer` aborts, which also terminates the `Producer`.

Before the `consume` calls, all necessary invocations to establish the `wait` calls, which can be seen as a monitor if the object is still alive, are shown. After that we see that two inputs from the `Producer` are sent to the `Consumer`, where the `wait` calls are still pending. In the end, the `Consumer`, and in consequence also the `wait` call, terminate. The termination leads to the retrieval of the exit reason (in form of an error) by the `Producer` from the associated future, which results in its termination as well.

One of the current limitations of this design is that due to the lack of subclassing, the boiler-plate implementation of the methods `wait` and `waitOn` in any class needs to be replicated (such as `Producer` and `Consumer` above). ABS offers so-called *deltas* to support assembly of *software product lines*. Although this feature can be used here in principle to inject code into a class, according to the current syntax of deltas, the method bodies would still have to be replicated in *each* delta. A potential improvement would be an extension of ABS which would allow injecting code into all classes implementing a particular interface.

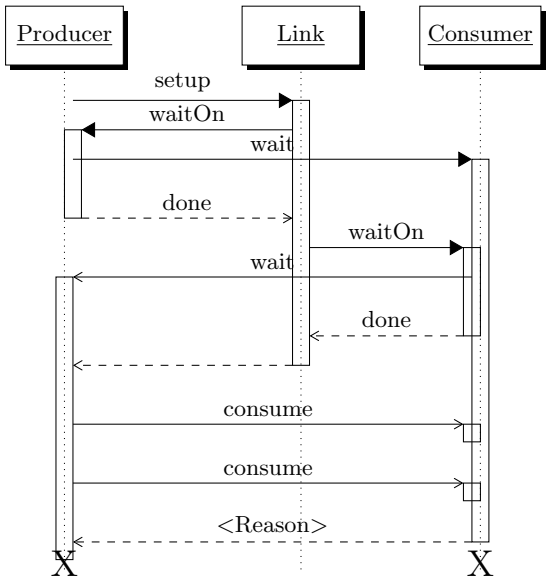

Fig. 5. Asynchronous calls in the Producer-Consumer example

Such functionality is well-known in *aspect-oriented programming*, and the ABS compiler should be easy to extend with a similar feature.

4 Operational Semantics and Application

A complete operational semantics of the core ABS language can be found in [11]. This section presents an operational semantics of the new language elements discussed in Section 3, omitting or simplifying parts that are not necessary for understanding the new error model. Figure 6 presents the runtime syntax of the language, while Figure 7 contains the operational semantics rules for the new rollback behavior, **abort** and **die** statements, and error propagation via futures.

The runtime state is a collection cn of objects, futures and method invocations. Objects are denoted $o(a, a', p, q)$, where a is the object state, a' the safe state at the previous suspension point. Dead objects are represented by their identifier o only. Object and process states a are mappings from identifiers to values, p is the currently running process or the symbol **idle** (denoting an object not currently running any process), and q is the process queue. A process p is written as $\{a|s\}$ with a a mapping from local variable identifiers to values and s a statement list.

In Figure 7 we elide the step of reducing expressions to values – evaluation is standard and can be seen in [11]. The Suspend rule saves the current state a, while the Abort rule reinstates a saved state while also removing the current process and filling the future f with an error term. The Die rule deactivates

$$
\begin{array}{rl}
cn & ::= \epsilon \mid fut \mid object \mid invoc \mid cn\ cn \\
fut & ::= f \mid f(val) \\
object & ::= o(a, a', p, q) \mid o \\
process & ::= \{a \mid s\} \\
q & ::= \epsilon \mid process \mid q\ q \\
invoc & ::= m(o, f, \overline{v})
\end{array}
\qquad
\begin{array}{rl}
a & ::= T\ x\ v \mid a, a \\
p & ::= process \mid \textsf{idle} \\
val & ::= v \mid error \\
v & ::= o \mid f \mid data \\
error & ::= e(val)
\end{array}
$$

Fig. 6. Runtime syntax. Overall program state is a set cn of futures, objects and invocation messages. Literals v are object identifiers o, future identifiers f, and number and string literals *data*.

the object and fills all futures of the object's processes with an error term. The Dead-Call rule provides a default error term as the result of a method call to a dead object. The other rules show the behavior of normal execution for these cases.

4.1 Discussion

From an implementation perspective, we note that the rollback mechanism appears reasonably cheap, as only that part of the state of the current object needs to be duplicated which is actually modified. This is easy to implement since ABS does not have destructive modification of data structures.

How to make best use of the rollback-mechanism is still up to the developer. We note that compared to traditional exception handling, a single method essentially corresponds to a **try**-block, whereas the caller specifies through a **safeget** and a subsequent case-distinction the possible **catch**-blocks, or decides to propagate any exceptions through **get**.

4.2 Application: Supervision

In Erlang the idea to let processes observe each other was taken further by constructing trees, where so called *supervisors start, observe and restart* their child processes. Supervision is one of the very important concepts, which is part of Erlang's highly regarded error handling capabilities [19]. Plugging in a supervisor as child of another supervisor generates a tree structure, which describes a structural view on components of a system. This tree structure enables both restarting of faulty leaves and of larger subtrees in case of repeated errors in a subsystem. So a faulty system with a supervisor tries to restart larger and larger parts of the whole system until enough faulty state is discarded and it is able to continue its operation.

Supervision for concurrent objects. Through linking, we can now apply the concept of supervision to concurrent objects. This enables modeling of a statically typed supervision tree that maintains active objects.

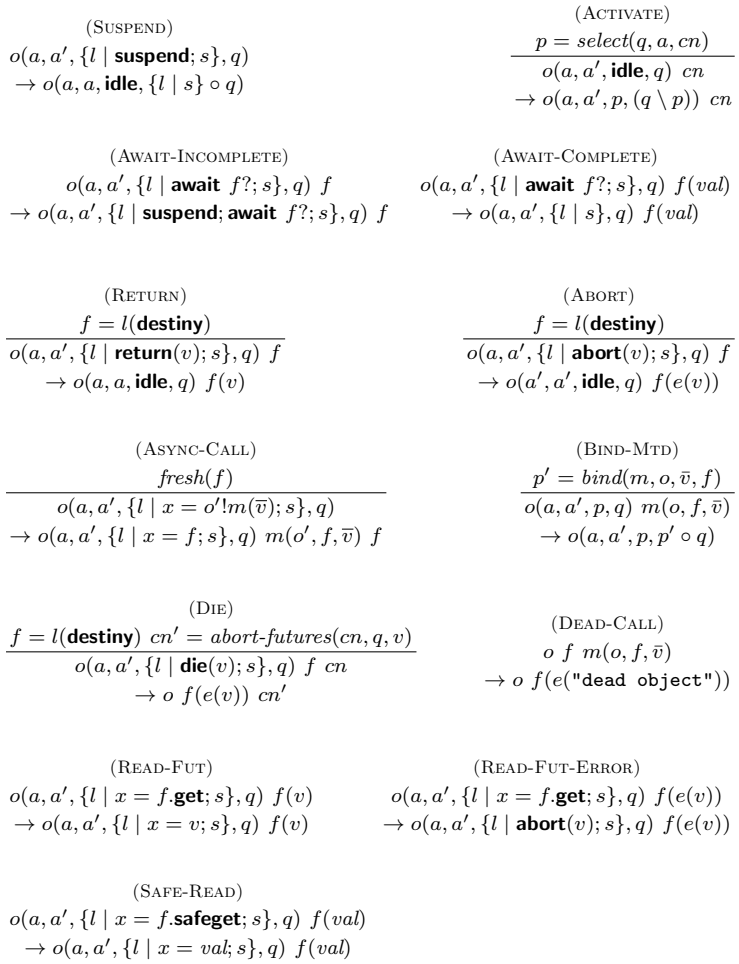

Fig. 7. Operational semantics. The following helper functions are assumed: *bind* creates a new process given a method name m, object o, arguments $\overline{v}$ and future f; *abort-futures* transforms a configuration, filling all futures f referenced from processes in queue q with an error term $e(v)$ while returning all other parts of the configuration unchanged; *select* chooses a process from a queue q that is ready to run.

```
1 Unit start(SupervisibleStarter child){
2  SupervisorLink sl=
3       new SupervisorLink(this,child);
4  Link l=new Link(sl,this);
5  await l!setup();
6  links=Cons(sl,links);
7  sl.start();
8 }
```

(a) Start a child

```
1 Unit died(SupervisibleStarter ss,
2         String error){
3  case strategy {
4   RestartAll => this.restart();
5   RestartOne => this.start(ss);
6   Prop => die error;
7  }
8 }
```

(b) Handle a deceased child

Fig. 8. Key methods of the `Supervisor`

To achieve a very generalized supervisor implementation we want to separate it from the concrete way of starting and linking children and want to be able to define different restart strategies. These strategies define the actions taken if a child terminates. Therefore we implemented a class `Supervisor` with following parameters: a list of `SupervisorStarter` objects, each of which specifies one child and implements the start and linking of this child; a strategy, which can be one of the following:

Restart one: Only the terminated child is restarted.
Restart all: If a child dies, it and all its siblings will be restarted.
Propagate: The supervisor and all children will terminate and the error will be thereby propagated to the next supervisor, ending at the root node of the runtime system.

This can be easily extended with other interesting strategies like rate limiting, e.g. propagating an error if a certain frequency of crashes is exceeded.

The implementation of the supervisor requires special considerations, as a supervisor has to start a list of children, keep track of them, has to detect a link failure and be able to forcefully terminate a child (for the *restart all* strategy). As the standard implementation of the link mechanism, shown in Figure 3, has on the error receiving side no indication about the source of the link error, every link to a child is represented by an object of class `SupervisorLink`. This object keeps the reference of the child specification (the `SupervisibleStarter` object) and passes it along to the `Supervisor`'s `died` method, which is depicted in Figure 8b. Furthermore this design allows one to forcefully kill one child, by terminating the associated `SupervisorLink`, which will—via linking—terminate the child.

For starting a child, a new `SupervisorLink` has to be created and linked to, so that in case the supervisor itself terminates (e.g. when the strategy is to propagate) all `SupervisorLink`s and children are terminated as well. This method is shown in Figure 8a.

5 Conclusion and Related Work

We have presented an extension to a concurrent object language, which incorporates automatic rollback to a "safe" (as conceptually defined by the developer

through a class invariant) state for the object that encountered an *abort*. Aborts either occur in the form of runtime errors, through an explicit call similarly to **throw**ing an exception, or from accessing a future which holds the result of an aborted computation.

The propagation- and detection mechanism for such faults allows us to model Erlang-like process linking, and the *safe* way of accessing futures corresponds roughly to exception handling with a distinction on the return result (normal return value vs. fault plus description).

We have implemented the proposed extension in a straight-forward manner in the prototypical (non-distributed) Erlang backend for ABS, and in the Maude simulator: The sources are publicly available in the ENVISAGE git repository at http://envisage-project.eu.

Related Work. Asynchronous computation with *futures* has been standardized in the Java API since Java SE 5 [10]. Due to the limitations of the so-called *generics* in the type system, no subtyping on futures is possible: this leads to the situation that (synchronous) method calls may make use of covariant return types, but for a type `B extending A`, a **Fut**`<B>` cannot be assigned to a **Fut**`<A>`. Our futures, based on ABS, do not have this limitation as futures stem from the functional data types and thus subtyping over parameterized types is safe due to the lack of destructive updates/writes. As first-class citizens, the ABS futures do not offer any cancellation and a process cannot affect another process except through sending messages (the Java API offers advisory cancellation, and—discouraged—forceful termination of threads).

Compared to Java futures, the ABS futures are intended to scale massively: while due to the limitations in Java's thread model only a restricted (by memory/stack requirements) number of threads can be effectively active (the standard reference [10] gives a limit in the "few thousands or tens of thousands", usually scheduled by an execution service); their intended use in ABS clearly follows Erlang's notion of virtually unbounded, light-weight, disposable threads.

A related failure model for an ABS-like language has also been discussed in [12]. To enable coordinated rollbacks, *compensations* are attached to method *returns*, in case a later condition indicates that a rollback across method calls should be necessary. The authors illustrate however that the distributed nature of compensation still does not make it easier to maintain *distributed invariants* involving several objects. Rollbacks in a concurrent system and their intricacies have also been discussed in the context of a higher-order π-calculus by Lanese et al. [16]. The entire design space of fault handling in a loosely coupled system is discussed in [15], but focuses on a more traditional approach of exception handlers to give developers an explicit means of recovery, instead of the implicit rollbacks presented here.

Unlike Java Card's transactions [6] our extension does not allow selective *non-atomic* updates, where a persistent value is modified within a transaction and *not* rolled back with the transaction. Our implementations do not store the entire heap upon method activation, but only the state of the current object.

A corresponding proof-theory as developed by Mostowski [17] for JAVA CARD-support in the KeY system should likewise be feasible for our approach.

Future work: The current rollback mechanism should also be easy to extend to *transactions* through a combination of versioning the object state and speculative execution. Also, rollbacks for a group of objects should at least semantically be easy to model, yet maintaining object graphs as additional state may make this approach too costly: every method invocation on another object would make this object a member of the transaction, and all objects would have to reach release points simultaneously to commit. Additionally, a distributed implementation of checking for such a commit would most likely be prohibitive. Instead of arbitrary object groups derived again following the discussion in [15], one may instead take advantage of so-called *concurrent object groups* (which are already present in ABS, but not discussed in this paper). They are used in ABS to model groups of objects running e.g. on the same node or hardware. Because of the intentionally tight coupling, one consideration is that a **die**-statement may even have the consequence of terminating the processes of an entire group, instead of the limited effect on the local object only that we discussed here.

Although the asynchronous communication mechanism together with the introduced failure mechanisms allows us to describe the communication behavior in a distributed system, the current semantics treats all calls—whether remote or local—the same. While this location transparency is also a feature of the Erlang language, it would be useful to reflect the topology of the system and resource aspects (such as processing power and communication latency) of the different nodes in a model. To this end, in [14] *deployment components* were introduced, which give the modeler the possibility to specify where objects are created and consequently where their processes run. Note that in contrast to Erlang, *objects* are allocated at creation-time, whereas Erlang allocates processes. On top of deployment components, resource costs and capabilities can be modeled and execution times can be estimated under different resource and deployment models. Simulation can then be used to examine the behavior of the (distributed) system wrt. artificially injected faults and deadline misses.

With respect to the supervision trees, we note that in the Erlang community, since the tree structure is specified through code, there was an interest in reverse-engineering the actual hierarchy for purposes of static analysis from the source code [18]. We hope that for top-down development, specification of the hierarchy can be made independent of the code, and is conversely more amenable to verification.

References

1. J. Armstrong. Erlang—a survey of the language and its industrial applications. In *Proc. INAP*, volume 96, 1996.
2. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

3. J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 9(1):29–43, 2013.
4. G. Candea and A. Fox. Crash-only software. In M. B. Jones, editor, *HotOS*, pages 67–72. USENIX, 2003.
5. D. Caromel and L. Henrio. *A Theory of Distributed Objects*. Springer, 2005.
6. Z. Chen. *Java Card Technology for Smart Cards*. Addison-Wesley, 2000.
7. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.
8. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 316–330. Springer, 2007.
9. C. C. Din, J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. *Journal of Logic and Algebraic Programming*, 81(3):227–256, 2012.
10. B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.
11. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th Intl. Symp. on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 142–164. Springer, 2011.
12. E. B. Johnsen, I. Lanese, and G. Zavattaro. Fault in the future. In W. D. Meuter and G.-C. Roman, editors, *COORDINATION*, volume 6721 of *LNCS*, pages 1–15. Springer, 2011.
13. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
14. E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Modeling application-level management of virtualized resources in ABS. In *Proc. 10th Intl. Symp. on Formal Methods for Components and Objects (FMCO 2011)*, volume 7542 of *LNCS*, pages 89–108. Springer, 2013.
15. I. Lanese, M. Lienhardt, M. Bravetti, E. B. Johnsen, R. Schlatte, V. Stolz, and G. Zavattaro. Fault model design space for cooperative concurrency. In *Submitted to ISoLA'14*, 2014.
16. I. Lanese, C. A. Mezzina, A. Schmitt, and J.-B. Stefani. Controlling reversibility in higher-order Pi. In J.-P. Katoen and B. König, editors, *CONCUR*, volume 6901 of *LNCS*, pages 297–311. Springer, 2011.
17. W. Mostowski. Formal reasoning about non-atomic Java Card methods in dynamic logic. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM*, volume 4085 of *LNCS*, pages 444–459. Springer, 2006.
18. J. Nyström and B. Jonsson. Extracting the process structure of Erlang applications. In *Erlang Workshop, Florence, Italy*, Sept. 2002. http://www.erlang.org/workshop/nystrom.ps.
19. S. Vinoski. Reliability with Erlang. *IEEE Internet Computing*, 11(6):79–81, 2007.

Appendix B

ABS Documentation

ABS Documentation

ABS Development Team – 1.0-SNAPSHOT

Table of Contents

Introduction

1. Abstract Behavioural Specification Language

The **ABS language** is a class-based object-oriented language that features algebraic data types and side effect-free functions. Syntactically, the ABS language tries to be as close as possible to the Java language, so that programmers that are used to Java can easily use the ABS language without much learning effort.

ABS offers programmers several features such as asynchronous method calls, futures to control these calls, interfaces for encapsulation and cooperative scheduling of method invocations inside concurrent (active) objects. Specifically any objectcreated in ABS represents an actor with encapsulated data. Similar to JAVA, their behaviour and state is defined by implementing interfaces with their corresponding methods. Thus they interact by making asynchronous calls to these methods which generate messages that are pushed into a queue specific to each actor. An actor progresses by taking a message out of its queue and processing it by executing its corresponding method. This feature combination results in a concurrent object-oriented model which is inherently compositional. The simplicity of ABS results from the fact that each actor is viewed as a separate processor making it very suitable for modeling distributed applications similar to MPI, with the added benefit of specifying a distinct behaviour for each actor without the connectivity issue.

Getting Started

2. Lexical Structure

This section describes the lexical structure of the ABS language. ABS programs are written in Unicode.

Line Terminators and White Spaces

Line terminators and white spaces are defined as in Java.

Comments

Comments are code fragments that are completely ignored and have no semantics in the ABS language. ABS supports two styles of comments: end-of-line comments and traditional comments.

End-Of-Line Comments

An end-of-line comment is a code fragment that starts with two slashes, e.g., `//` text. All text that follows `//` until the end of the line is treated as a comment.

Syntax

```
// this is a comment
module A; // this is also a comment
```

Traditional Comments

A traditional comment is a code fragment that is enclosed in `/* */`, e.g., `/* this is a comment */`. Nested traditional comments are not possible.

```
/* this
is a multiline
comment */
```

Identifiers

ABS distinguishes identifier and type identifier. They differ in the first character, which must be a lower-case character for identifiers and an upper-case character for type identifiers.

Keywords

The following words are keywords in the ABS language and are not regarded as identifiers.

adds	core	export	hasMethod	let	product	this
after	class	features	if	local	productline	type
assert	data	from	implements	modifies	removes	when
await	def	get	import	module	return	while
builtin	delta	hasField	in	new	skip	case

Literals

A literal is a textual representation of a value. ABS supports three kinds of literals, integer literals, string literals, and the null literal.

Separators

The following characters are separators:

```
(     )     {     }     [     ]     \           \;
```

Operators

The following tokens are operators:

```
||   &&   ==   !=   <   >   <=   >=   +   -   *   /   %   ~   &
```

---

3. Names and Types

Names

A name in ABS can either be a simple identifier as described above, or can be qualified with a type name, which represents a module. Examples for syntactically valid names are: head, x, ABS.StdLib.tail. Examples for type names are: Unit, X, ABS.StdLib.Map.

Types

Types in ABS are either plain type names or can have type arguments. A type name can refer to a data type, an interface, a type synonym, and a type parameter. Note that classes cannot be used as types in ABS. In addition, only parametric data types can have type arguments. Examples for syntactically valid types are: Bool, ABS.StdLib.Int, List<Bool>, ABS.StdLib.Map<Int,Bool>.

Type Synonyms

Type Synonyms define synonyms for otherwise defined types. Type synonyms start with an uppercase letter.

Syntax

```
type Filename = String
type Filenames = Set<Filename>
type Servername = String
type Packet = String
type File = List<Packet>
type Catalog = List<Pair<Servername,Filenames>>
```

3.1. Built-in Types

The most basic of built-in types is the Int type, which represents integers of arbitrary size. Values of type Int can be constructed by using integer literals or arithmetic expressions:

```
0  has type Int
1  has type Int
3  has type Int
3+1  has type Int
```

A related type is the Rat for representing rational numbers. Rational values are obtained via the division (/) operator and have arbitrary precision. An example of such values:

```
1/4   has type Rat
5/2/4 has type Rat
```

The Int type is a subtype of Rat; this means that Int expressions are automatically converted to Rat expressions, whenever a Rat type is expected. Some examples of Rat expressions:

```
0 can also have type Rat
1 can also have type Rat
4+3 can also have the Rat
```

The adverse does not work. The user that wants to convert from Rat to Int types have to explicitly use the truncate(number) function.

The String built-in type represents String values, constructed either by string literals or by string-specific operators coming from the ABS standard library. Examples of Strings:

```
"hello world\n"  is a string literal
"standard" + "library" is a string expression (concatenation)
```

According to other functional languages, Strings in ABS are immutable data. Desimilar to other function languages, ABS Strings are not represented as list of characters; instead they have a hidden-to-the-user, efficient, internal representation.

The Fut type is a special built-in type to signal an ABS value that will become available (evaluated) in the future. Fut is a so-called parameterized type, written as Fut<T>, where T is its type parameter. The value that a future holds and will return can be of any concrete type, as in the example:

```
Fut<String> is the type that will return a string
Fut<List<Rat>> is the type that will return a list of rational numbers
```

3.2. Algebraic Data Types

Algebraic Data Types make it possible to describe data in an immutable way. In contrast to objects, data types do not have an identity and cannot be mutated. This makes reasoning about data types much simpler than about objects. Data types are built by using Data Type Constructors (or constructors for short), which describe the possible values of a data type.

Syntax

```
data IntList = NoInt | Cons(Int, IntList);
data Bool = True | False;
```

Parametric Data Types

Parametric Data Types are useful to define general-purpose data types, such as

lists, sets or maps. Parametric data types are declared like normal data types but have an additional type parameter section inside broken brackets (< >) after the data type name.

```
data List<A> = Nil | Cons(A, List<A>);
```

Predefined Algebraic Data Types

The following Algebraic Data Types are predefined and come bundled with the ABS standard library:

- data Bool = True | False;++ The boolean type with constructors True and False and the usual Boolean infix and prefix operators.
- data Unit = Unit;++ The unit type with only one constructor Unit (for methods without return values).
- data Int; An arbitrary integer (Z) for which values are constructed by using integer literals and arithmetic expressions.
- data Rat;. A rational number (Q). Rational values are obtained via the division (/) operator and have arbitrary precision. Assigning rational values to variables of type Int, either explicitly or implicitly by passing them to a function or method expecting an integer, rounds towards zero.
- data String;. A string for which values are constructed by using string literals and operators.
- data Fut<T>;. Representing a future. A future cannot be explicitly constructed, but it is the result of an asynchronous method call. The value of a future can only be obtained by using the get expression.
- data List<A> = Nil | Cons(A, List<A>)++ Where Nil is the empty list, and Cons appends an element of type A in the front of the list.

N-ary Constructors

For data types of arbitrary size, like lists, maps and sets, it is undesirable having to write them down in the form of nested constructor expressions. For this purpose, ABS provides a special syntax for nary constructors, which are transformed into constructor expressions via a user-supplied function.·

Syntax

```
def Set<A> set<A>(List<A> l) = case l {
    Nil => EmptySet;
    Cons(hd, tl) => Insert(hd, set(tl));
    };

{
Set<Int> s = set[1, 2, 3];
}
```

An expression type[parameters*] is transformed into a literal by handing it to a function named type which takes one parameter of type List and returns an expression of type Type. (It is desirable, although not currently enforced, that type and Type are the same word, just with different capitalization.)

Abstract Data Types

Using the module system it is possible to define abstract data types. For an abstract data type, only the functions that operate on them are known to the client, but not its constructors. This can be easily realized in ABS by putting such a data type in its own module and by only exporting the data type and its functions, without exporting the constructors.

3.3. The Exception Type

In higher-level programming languages, exceptions are generally used to signal an *erroneous* or *abnormal* runtime behaviour of the program, that should be treated (handled) separately compared to normal values.

The Exception type is a special built-in data type that looks similar to an Algebraic Data Type (immutable, no identity) but with a notable difference: the exception data type **can** be extended with new (user-provided) data constructors. Based on this fact, the user has the ability to, besides using the predefined exceptions of the ABS standard library, write arbitrary exceptions specific to the user's program.

To define a new exception (data constructor) the user has to write:

```
exception MyException;
```

An exception can also take any number of arguments as:

```
exception AnotherException(Int, String, Bool);
```

In ABS, exceptions are first-class citizens of the language; the user can construct exception-values, assign them to variables or pass them in expressions. All these exception-values are typed by the type Exception . However, an exception-value can only acquire the special meaning of abnormal behaviour when the user explicitly says so with a throw keyword. We will visit the throw keyword together with how to recover from exceptions (catch keyword) in a later section.

Predefined exceptions in the Standard Library

DivisionByZeroException

Raised in arithmetic expressions when the divisor (denominator) is equal to 0, as in 3/0

AssertionFaiException

The assert keyword was called with False as argument

PatternMatchFailException

The pattern matching was not complete. In other words all c catch-all clause

NullPointerException

A method was called on a null object

StackOverflowException

The calling stack has reached its limit (system error)

HeapOverflowException

The memory heap is full (system error)

KeyboardInterruptException

The user pressed a key sequence to interrupt the running ABS program

3.4. Interface Types

Interfaces in ABS are similar to interfaces in Java. Unlike Java, objects in ABS are typed exclusively by interfaces, and **not** classes.

To introduce an interface:

```
interface Animal {
...
}
```

Interfaces can be extended from (multiple) base interfaces:

```
interface Bird extends Animal, Flying {
...
}
```

Let's consider the example of an object that represents a "seagull". Such*seagull* object can have either the type of a Bird, Animal or Flying, depending on the object's particular usage in the program. In terms of type theory, this feature is called *nominal subtyping*. An example of well-typed expressions that make use of Interface types:

```
seagull                     can be typed by: Bird or Animal or Flying
list[seagull, bee]          can be typed by: Animal or Flying
set[seagull, bee, boeing]   can be typed by: Flying
```

Functional Sublanguage

4. Pure Expressions

Pure Expressions are side effect-free expressions. This means that these expressions cannot mod- ify the heap.

Let Expressions

These expressions bind variable names to pure expressions.

Syntax

```
let (Bool x) = True in ~x
```

Data Type Constructor Expressions

They are expressions that create data type values by using data type constructors. Note that for data type constructors that have no parameters, the parentheses are optional.

Syntax

```
True
Cons(True, Nil)
ABS.StdLib.Nil
```

Function Applications

Function Applications apply functions to arguments.

Syntax

```
tail(Cons(True, Nil))
ABS.StdLib.head(list)
```

If-Then-Else Expression

ABS has a standard if-then-else expression.

Syntax

```
if 5 == 4 then True else False
```

Case Expressions / Pattern Matching

ABS supports pattern matching by the Case Expression. It takes an expression as first argument, which a series of patterns is matched against. The value of the case expression itself is the value of the expression on the right-hand side of the first matching expression. It is an error if no pattern matches the expression.

4.1. Patterns

There are five different kinds of patterns available in ABS:

- Pattern Variables (e.g., x, where x is not bound yet)
- Bound Variables (e.g., x, where x is bound)
- Literal Patterns (e.g., 5)
- Data Constructor Patterns (e.g., Cons(Nil,x))
- Underscore Pattern (_)

Pattern Variables

Pattern variables are simply unbound variables. Like the underscore pattern, these variables match every value, but, in addition, bind the variable to the matched value. The bound variable can then be used in the right-hand-side

expression of the corresponding branch. Typically, pattern variables are used inside of data constructor patterns to extract values from data constructors. For example

```
def A fromJust<A>(Maybe<A> a) =
case a {
 Just(x) => x;
};
```

Bound Variables

If a bound variable is used as a pattern, the pattern matches if the value of the case expression is equal to the value of the bound variable.

```
def Bool contains<A>(List<A> list, A value) =
case list {
 Nil => False;
 Cons(value, _) => True;
 Cons(_, rest) => contains(rest, value);
};
```

Literal Patterns

Literals can be used as patterns. This is similar to bound variables, because the pattern matches if the value of the case expression is equal to the literal value.

```
def Bool isEmpty(String s) =
case b {
 "" => True;
 _ => False;
};
```

Data Constructor Patterns

A data constructor pattern is like a standard data constructor expression, but where certain sub expressions can be patterns again.

```
def Bool negate(Bool b) =
case b {
 True => False;
 False => True;
};
```

```
def List<A> remainder(List<A> list) =
case b {
 Cons(_, rest) => rest;
};
```

Underscore Pattern

The underscore pattern (_) simply matches every value. It is generally used as the last pattern in a case expression to define a default case. For example:

```
def Bool isNil<A>(List<A> list) =
case list {
 Nil => True;
 _ => False;
};
```

4.2. Type Checking

A case expression is type-correct if and only if all its expressions and all its branches are type-correct and the right-hand side of all branches have a common super type. This common super type is also the type of the overall case expression. A branch (a pattern and its expression) is type-correct if its pattern and its right-hand side expression are type-correct. A pattern is type-correct if it can match the corresponding case ex-pression.

Operator Expressions

ABS has a number of predefined operators which can be used to form Operator Expressions.

The following table describes the meaning as well as the associativity and the precedence of the different operators. They are grouped according to precedence, as indicated by horizontal rules, from low precedence to high precedence.

Expression	Meaning	Associativity	Argument types	Result type
e1 || e2	logical or	left	Bool, Bool	Bool
e1 && e2	logical and	left	Bool, Bool	Bool
- e	integer negation	right	number	number

Expression	Meaning	Associativity	Argument types	Result type
e1 == e2	equality	left	compatible	Bool
e1 != e2	inequality	left	compatible	Bool
e1 < e2	less than	left	number, number	Bool
e1 ⇐ e2	less than or equal to	left	number, number	Bool
e1 > e2	greater than	left	number, number	Bool
e1 >= e2	greater than or equal to	left	number, number	Bool
e1 + e2	concatenation	left	String, String	String
e1 + e2	addition	left	number, number	number
e1 - e2	subtraction	left	number, number	number
e1 * e2	multiplication	left	number, number	number
e1 / e2	division	left	number, number	Rat
e1 % e2	modulo	left	number, number	Int
~ e	logical negation	right	Bool	Bool
- e	integer negation	right	number	number

5. Functions

Functions in ABS define names for parametrized data expressions. A Function in ABS is always side effect-free, which means that it cannot manipulate the heap.

Syntax

```
def Int length(IntList list) =
case list {
 Nil => 0;
 Cons(n, ls) => 1 + length(ls);
};
```

Parametric Functions

Parametric Functions allow to work with parametric data types in a general way. For exam- ple, given a list of any type, a parametric function head can return the first element, regardless of its type. Parametric functions are defined like normal functions but have an additional type parameter section inside angle brackets (< >) after the function name.

Syntax

```
def A head<A>(List<A> list) =
 case list {
 Cons(x, xs) => x;
};
```

(Note that head is a partial function.)

Object Model

6. Interfaces

Interfaces in ABS are similar to interfaces in Java. They have a name, which defines a nominal type, and they can extend arbitrary many other interfaces. The interface body consists of a list of method signature declarations. Method names start with a lowercase letter.

The interfaces in the example below represent a database system, providing functionality to store and retrieve files, and a node of a peer-to-peer file sharing

system. Each node of a peer-to-peer system plays both the role of a server and a client.

Syntax

```
interface DB {
 File getFile(Filename fId);
 Int getLength(Filename fId);
 Unit storeFile(Filename fId, File file);
 Filenames listFiles();
}
interface Client {
 List<Pair<Server,Filenames>> availFiles(List<Server> sList);

 Unit reqFile(Server sId, Filename fId);
}
interface Server {
 Filenames inquire();
 Int getLength(Filename fId);
 Packet getPack(Filename fId, Int pNbr);
}
interface Peer extends Client, Server {
 List<Server> getNeighbors();
}
```

---

7. Classes

Like in typical class-based languages, classes in ABS are used to create objects. Classes can implement an arbitrary number of interfaces. Classes do not have constructors in ABS but instead have class parameters and an optional init block. Class parameters actually define additional fields of the class that can be used like any other declared field.

Syntax

```
class DataBase(Map<Filename,File> db) implements DB {
 File getFile(Filename fId) {
  return lookup(db, fId);
 }

 Int getLength(Filename fId){
  return length(lookup(db, fId));
 }

 Unit storeFile(Filename fId, File file) {
  db = insert(Pair(fId,file), db);
 }

 Filenames listFiles() {
  return keys(db);
 }

 }

class Node(DB db, Peer admin, Filename file) implements Peer {

 Catalog catalog;
 List<Server> myNeighbors;
 // implementation...

}
```

Active Classes

A class can be active or passive. Active classes start an activity on their own upon creation. Passive classes only react to incoming method calls. A class is active if and only if it has a run method:

```
Unit run() {
 // active behavior ...
}
```

The run method is called after object initialization.

---

8. Deployment Components

A Deployment Component (abbreviated as `DC`) is the abstraction of a computational unit which is responsible for running ABS computations (programs). Such a computational unit can either be realised by an OS process, physical machine, virtual machine or a multitude of machines. By using DCs, the programmer can write ABS programs that span across multiple computational

units, similar to a distributed setting.

Based on the fact that a DC is a single **unit** that (pro)**actively** behaves (runs ABS code), it is modelled as an *active object*, discussed in the previous section. All DC objects are typed by the `DC` interface, defined in ABS as:

```
interface IDC {
  Unit shutdown();
  Triple<Rat,Rat,Rat> load();
}
```

With the `shutdown` method, a DC can safely be brought down, subsequently freeing its occupied resources. The `load` method permits the user to probe for the average load of the computational unit, that is how much utilized (busy) was the unit in the last 1 , 5 and 15 minutes of execution. The following example is self-explanatory:

```
DC dc1 = new MyDC();
Fut<Triple<Rat,Rat,Rat>> f_avgs = dc1 ! load();
Triple<Rat,Rat,Rat> avgs = f_avgs.get();
dc1 ! shutdown();
```

The ABS language specification does not define any built-in DC classes. It is in the discretion of the ABS backends to provide with suitable DC classes (implementations).

After a new DC object is created, its associated computational unit is started and sits waiting to execute ABS computations. To start ABS computations, the user must create (active) objects inside the remote computational unit. This is called `object spawning`, illustrated by the example:

```
Interf1 o1 = dc1 spawns Class1(params..);
o1 ! method1( params ..);
this.method2( o1 );
```

The `spawns` keyword creates a new object in the given DC. It behaves similar to the `new` keyword, in the sense that the created object will be also placed in a new COG. The returned object reference (`o1` in the example) is treated as normal (can be passed to arguments, called for its methods, etc.).

The keyword `thisDC` is provided that can be put anywhere inside an ABS program to return the computational unit where the calling code executes in.

```
{
  DC whereami = thisDC;
}
```

Imperative Sublanguage

9. Control Statements

Similar to Java, a statement is each of the individual instructions of a program, like the variable declarations and expressions seen in previous sections. They always end with a semicolon (;), and are executed in the same order in which they appear in a program. ABS provides flow control statements that serve to specify what has to be done by our program, when, and under which circumstances. Statements in ABS are not evaluated to a value. If one wants to assign a value to statements it would be the **Unit** value.

Block

This statement is also known as a**compound statement** and consists of a group of statement grouped together defining a name scope for variables.

```
{
 a = a + 1;
 n = n % 10;
}
```

Selection statement

```
if (5 < x) {
 y = 6;
}
else {
 y = 7;
}
if (True)
 x = 5;
```

Iteration statement (Loop)

```
while (x < 5)
 x = x + 1;
```

Variable Declaration Statements

A variable declaration statement is used to declare variables. A variable has an optional initialization expression for defining the initial value of the variable. The initialization expression is mandatory for variables of data types. It can be left out only for variables of reference types, in which case the variable is initialized with null.

```
Bool b = True;
```

Assign Statement

The **Assign Statement** assigns a value to a variable or a field.

```
this.f = True;
x = 5;
```

Await Statement

Await Statements suspend the current task until the given guard is true [7]. The task will not be suspended if the guard is already initially true. While the task is suspended, other tasks within the same COG can be activated. Await statements are also called scheduling points, because they are the only source positions, where a task may become suspended and other tasks of the same COG can be activated.

```
Fut<Bool> f = x!m();
await f?;
await this.x == True;
await f? & this.y > 5;
```

Suspend Statement

A Suspend Statement causes the current task to be suspended.

```
suspend;
```

Assert Statement

An **Assert Statement** is a statement for asserting certain conditions.

```
assert x != null;
```

Return Statement

A **Return Statement** defines the return value of a method. A return statement can only appear as a last statement in a method body.

```
return x;
```

Case Statement

The case statement, like the case expression, takes an expression as first argument, which is matched against a series of patterns. The effect of executing the case statement is the execution of the statement (which can be a block) of the first branch whose pattern matches the expression. An example follows:

```
Pair<Int, Int> p = Pair(2, 3);
Int x = 0;
case p {
  Pair(2, y) => { x = y; skip; }
  _ => x = -1;
}
```

Exception-signaling statement

The keyword-statement `throw` is used to signal exceptions (runtime errors). It takes a single argument which is the exception-value to throw. For example:

```
{
 Int x = -1;
 if (x<0) {
    throw NegativeNumberException(x);
 }
 else {
   if (x==0) {
      throw ZeroNumberException;
   }
   else ...
}
```

> The 'throw' statement can only be used inside imperative code. Throwing user-exceptions inside functional code is considered bad practice: the user's function must be written instead to return an Either<Exception, A> value, as in the example:

```
def Either<Exception, Int> f(x,y) = if (y < 0)
                                    then Left(NegativeNumberException)
                                    else Right(...)
```

Despite this, there are certain built-in system-exceptions (see Section 3.3) that can originate from erroneous functional code. Examples of these are `DivisionByZeroException` and `PatternMatchFailException`, implicitly signaled by the ABS system.

When an exception is raised (signaled), the normal flow of the program will be abrupted. In order to resume the normal flow, the user has to explicitly **handle** the exception.

Exception-handling Statement

To handle an exception --- either explicitly signaled using the `throw` keyword or implicitly by a system exception --- the user has to surround the offending code with a `try` block. The statements in the try block will be executed in sequence until an exception happens. Upon an exception, the execution of the try block will stop and the exception will be matched against the exception-patterns defined in the `catch` block.

The catch block behaves similar to the `case statement`, with the only difference that the patterns can only have the type `Exception`. When the exception-pattern is matched, the statements associated with its catch clause will be executed.

After defining the catch block, the user can *optionally* supply a `finally` block of statements, that will be executed regardless of an exception happening or not.

The syntax is the following:

```
try {
 stmt1;
 stmt2;
 ....
}
catch {
  exception_pattern1  => stmt_or_block;
  exception_pattern2  => ... ;
  ...
  _ => ...
}
finally {
  stmt3;
  stmt4;
}
```

If there are no matching catch-clauses, the finally block will first be accordingly executed, before re-throwing the exception to its parent caller. Conversely, if the parent caller does not (correctly) handle the re-thrown exception, the exception will be propagated to its own parent caller, and so forth and so on.

Expression Statement

An **Expression Statement** is a statement that only consists of a single expression. Such statements are only executed for the effect of the expression.

```
new C(x);
```

---

10. Expressions with Side Effects

Beside pure expressions, ABS has expressions with side effects. However, these expressions are defined in such a way that they can only have a single side effect. This means that subexpressions of expressions can only be pure expressions again. This restriction simplifies the reasoning about ABS expressions.

New Expression

A New Expression creates a new object from a class name and a list of arguments. In ABS objects can be created in two different ways. Either they are created in the current COG, using the standard new local expression, or they are created in a new COG by using the new expression.

Syntax

```
new local Foo(5)
new Bar()
```

Standard Object Creation

When using the new local expression, the new object is created in the current COG, i.e., the COG of the current receiver object.

COG Object Creation

The concurrency model of ABS is based on the notion of COGs [?]. An ABS system at runtime is a set of concurrently running COGs. A COGs can be seen as an isolated subsystem, which has its own state (an object-heap) and its own internal behavior. COGs are created implicitly when creating a new object by using the new expression.

Synchronous Call Expression

A Synchronous Call consists of a target expression, a method name, and a list of argument expressions.

```
Bool b = x.m(5);
```

Asynchronous Call Expression

An Asynchronous Call consists of a target expression, a method name, and a list of argument expressions. Instead of directly invoking the method, an asynchronous method call creates a new task in the target COG, which is executed asynchronously. This means that the calling task proceeds independently after the call, without waiting for the result. The result of an asynchronous method call is a future (Fut<V>), which can be used by the calling task to later obtain the result of the method call. That future is resolved by the task that has been created in the target COG to execute the method.

```
Fut<Bool> f = x!m(5);
```

Get Expression

A Get Expression is used to obtain the value from a future. The current task is blocked until the value of the future is available, i.e., until the future has been resolved. No other task in the COG can be activated in the meantime.

```
Bool b = f.get;
```

Await Expression

A common pattern for asynchronous calls is:

- Execute an asynchronous call expression, store the future in a variable
- `await` on the future
- Assign the result to a variable

```
Fut<A> fx = o!m();
await fx?;
A x = fx.get;
```

The await expression is a shorthand for this pattern. The preceding example can be written as follows, without the need to introduce a name for the future:

```
A x = await o!m();
```

Modules

11. Defining a Module

For name spacing, code structuring, and code hiding purposes, ABS offers a module system. The module system of ABS is very similar to that of Haskell. It uses, however, a different syntax that is similar to that of Java and Python.

```
ModuleDecl  ::= 'module' TypeName ';' [ExportList] [ImportList] Decl* [Block]
ExportList  ::= Export*
ImportList  ::= Import*
Export      ::= 'export' AnyNameList ['from' TypeName] ';'
              | 'export' '*' ['from' TypeName] ';'
Import      ::= 'import' AnyNameList ['from' TypeName] ';'
              | 'import' '*' 'from' TypeName ;
AnyNameList ::= AnyName [, AnyName]
AnyName     ::= Name | TypeName

Decl        ::= FunDecl | TypeSynDecl | DataTypeDecl | InterfaceDecl | ClassDecl
```

A module with name MyModule is declared by writing

```
module MyModule;
```

This declaration introduces a new module name `MyModule` which can be used to qualify names. All declarations which follow this statement belong to the module `MyModule`. A module name is a type name and must always start with an upper case letter.

The module `ABS.StdLib` contains the standard library and is automatically imported by every module.

---

12. Exporting Identifiers

By default, modules do not export any names. In order to make names of a module usable to other modules, the names have to be *exported*. For example, to export a data type and a data constructor, one can write something like this:

```
module Drinks;
export Drink, Milk;
data Drink = Milk | Water;
```

Note that in this example, the data constructor `Water` is not exported, and can thus not be used by other modules. By only exporting the data type without any of its constructors, one can realize *abstract data types* in ABS.

A special export clause `export *;` exports all names that are *defined* in the module. In particular, this means that imported names are *not* exported (but can be re-exported via additional `export` clauses).

```
export *;
```

---

13. Importing Identifiers

In order to use exported names of a module in another module, the names have to be imported. In a module definition, an optional list of import clauses follows

the list of export clauses. For example, to write a module that imports the `Drink` data type of the module `Drinks` one can write:

```
module Bar;
import Drinks.Drink;
```

After a name has been imported, it can be used inside the module in a fully qualified way.

To use a name from another module in an unqualified way requires an *unqualified import*. For example, to use the `Milk` data constructor inside the `Bar` module, without having to qualify it with the `Drinks` module each time, the following unqualified import statement is used:

```
module Bar;
import Milk from Drinks;
```

(Note that this kind of import also imports the qualified names.) In this example, the names `Milk` and `Drinks.Milk` can be used inside the module `Bar`.

To use all exported names from another module in an unqualified way one can write:

```
import * from SomeModule;
```

13.1. Exporting Imported Names

It is possible to export names that have been imported. For example,

```
module Bar;
export Drink;
import * from Drinks;
```

exports data type `Drink` that has been imported from `Drinks`.

To export all names imported from a certain module one can write

```
export * from SomeModule;
```

In this case, all names that have been imported from module `SomeModule` are

exported. For example,

```
module Bar;
export * from Drinks;
import * from Drinks;
```

exports all names that are exported by module `Drinks`.

However, in this example:

```
module Bar;
export * from Drinks;
import Drink from Drinks;
```

only `Drink` is exported as this is the only name imported from module `Drinks`. Note: only names that are visible in a module can be exported by that module.

To only export some names from a certain module one can write, for example:

```
module Bar;
export Drink from Drinks;
import * from Drinks;
```

This only exports `Drink` from module `Drinks`.

Feature Modelling

14. Deltas

ABS supports the delta-oriented programming model, an approach that aids the development of a set of programs simultaneously from a single code base, following the software product line engineering approach. In delta-oriented programming, features defined by a feature model are associated with code modules that describe modifications to a core program. In ABS, these modules are called *delta modules*. Hence the implementation of a software product line in ABS is divided into a *core* and a set of delta modules.

The core consists of a set of ABS modules that implement a complete software

product of the corresponding software product line. Delta modules (or *deltas* in short) describe how to change the core program to obtain new products. This includes adding new classes and interfaces, modifying existing ones, or even removing some classes from the core. Delta modules can also modify the functional entities of an ABS program, that is, they can add and modify data types and type synonyms, and add functions.

Deltas are applied to the core program by the ABS compiler front end. The choice of which delta modules to apply depends on the selection of a set of features, that is, a particular product of the SPL. The role of the ABS compiler front end is to translate textual ABS models into an internal representation and check the models for syntax and semantic errors. The role of the back ends is to generate code for the models targeting some suitable execution or simulation environment.

```
DeltaDecl      ::= 'delta' TypeId [DeltaParams] ';' [ModuleAccess] ModuleModifier*
ModuleModifier ::= 'adds' ClassDecl
                 | 'removes' 'class' TypeName ';'
                 | 'modifies' 'class' TypeName
                   ['adds' TypeId (',' TypeId)*] ['removes' TypeId (',' TypeId)*]
                   '{' Modifier* '}'
                 | 'adds' InterfaceDecl
                 | 'removes' 'interface' TypeName ';'
                 | 'modifies' 'interface' TypeName '{' InterfaceModifier* '}'
                 | 'adds' FunctionDecl
                 | 'adds' DataTypeDecl
                 | 'modifies' DataTypeDecl
                 | 'adds' TypeSynDecl
                 | 'modifies' TypeSynDecl
                 | 'adds' Import
                 | 'adds' Export

InterfaceModifier ::= 'adds' MethSig ';'
                    | 'removes' MethSig ';'

Modifier ::= 'adds' FieldDecl
           | 'removes' FieldDecl
           | 'adds' MethDecl
           | 'removes' MethSig
           | 'modifies' MethDecl

DeltaParams ::= '(' DeltaParam (',' DeltaParam)* ')'

DeltaParam ::= Identifier HasCondition*
             | Type Identifier

ModuleAccess ::= 'uses' TypeId ';'

HasCondition ::= 'hasField' FieldDecl
               | 'hasMethod' MethSig
               | 'hasInterface' TypeId
```

The `DeltaDecl` clause specifies the syntax of delta modules, consisting of a unique identifier, a module access directive, a list of parameters and a sequence of module modifiers. The *module access* directive gives the delta access to the namespace of a particular module. In other words, it specifies the ABS module to which modifications using unqualified identifiers apply by default. A delta can still make changes to several modules by fully qualifying the `TypeName` of module modifiers.

While delta modelling supports a broad range of ways to modify an ABS model, not all ABS program entities are modifiable. These unsupported modifications are listed here for completeness. While these modifications could be easily specified and implemented, we opted not to overload the language with features that have

not been regarded as necessary in practice:

Class parameters and init block

Deltas currently do not support the modification of class parameter lists or class init blocks.

Deltas

currently only support adding functions, and adding and modifying data types and type synonyms. Removal is not supported.

Modules

Deltas currently do not support adding new modules or removing modules.

Imports and Exports

While deltas do support the addition of import and export statements to modules, they do not support their modification or removal.

Main block

Deltas currently do not support the modification of the program's main block.

---

15. Feature Modelling

ABS provides language constructs and tools for modelling variable systems following Software Product Line (SPL) engineering practices.

Software variability is commonly expressed using features which can be present or absent from a product of the product line. Features are organised in a feature model, which is essentially a set of logical constraints expressing the dependencies between features. Thus the feature model defines a set of legal feature combinations, which represent the set of valid software products that can be built from the given features.

15.1. Specifying the Product Line

The ABS configuration language links feature models, which describe the structure of a SPL, to delta modules, which implement behaviour. The configuration defines, for each selection of features satisfied by the product selection, which delta modules should be applied to the core. Furthermore, it

guides the code generation by ordering the application of the delta modules.

```
Configuration ::= 'productline' TypeId ';' Features ';' DeltaClause*
Features      ::='features' FName (',' FName)*
DeltaClause   ::= 'delta' DeltaSpec [AfterCondition] [ApplicationCondition] ';'
DeltaSpec     ::= DeltaName ['(' DeltaParams ')']
DeltaName     ::= TypeId
DeltaParams   ::= DeltaParam (',' DeltaParam)*
DeltaParam    ::= FName | FName'.'AName
AfterClause   ::= 'after' DeltaName (',' DeltaName)*
WhenClause    ::= 'when' AppCond
AppCond       ::= AppCond '&&' AppCond
                | AppCond '||' AppCond
                | '~' AppCond
                | '(' AppCond ')'
                | FName
```

Features and delta modules are associated through *application conditions*, which are logical expressions over the set of features and attributes in a feature model. The collection of applicable delta modules is given by the application conditions that are true for a particular feature and attribute selection. By not associating the delta modules directly with features, a degree of flexibility is obtained.

Each delta clause has a `DeltaSpec`, specifying the name of a delta module name and, optionally, a list of parameters; an `AfterClause`, specifying the delta modules that the current delta must be applied after; and an application condition `AppCond`, specifying an arbitrary predicate over the feature names (`FName`) and attribute names (`AName`) in the feature model that describes when the given delta module is applied.

```
productline DeltaResourceExample;
features Cost, NoCost, NoDeploymentScenario, UnlimitedMachines, LimitedMachines,
Wordcount, Wordsearch;
delta DOccurrences when Wordsearch;
delta DFixedCost(Cost.cost) when Cost;
delta DUnboundedDeployment(UnlimitedMachines.capacity) when UnlimitedMachines;
delta DBoundedDeployment(LimitedMachines.capacity, LimitedMachines.machinelimit)
when LimitedMachines;
```

15.2. Specifying Products

ABS allows the developer to name products that are of particular interest, in order to easily refer to them later when the actual code needs to be generated. A product definition states which features are to be included in the product and sets attributes of those features to concrete values.

```
Selection ::= 'product' TypeId '(' FeatureSpecs ')' ';'
FeatureSpecs ::= FeatureSpec (',' FeatureSpec)*
FeatureSpec ::= FName [AttributeAssignments]
AttributeAssignments ::= '{' AttributeAssignment (',' AttributeAssignment '}'
AttributeAssignment ::= AName '=' Literal
```

Here are some product definitions for the `DeltaResourceExample` productline:

```
product WordcountModel (Wordcount, NoCost, NoDeploymentScenario);
product WordcountFull (Wordcount, Cost{cost=10}, UnlimitedMachines{capacity=20});
product WordsearchFull (Wordsearch, Cost{cost=10},
UnlimitedMachines{capacity=20});
product WordsearchDemo (Wordsearch, Cost{cost=10}, LimitedMachines{capacity=20,
machinelimit=2});
```

15.3. The Feature Model

The `FeatureModel` clause specifies a number of "orthogonal" root feature models along with a number of extensions that specify additional constraints, typically cross-tree dependencies. Its grammar is as follows:

```
FeatureModel ::= ('root' FeatureDecl)* FeatureExtension*
FeatureDecl  ::= FName [ '{' [Group] AttributeDecl* Constraint* '}' ]
FeatureExtension ::= 'extension' FName '{' AttributeDecl* Constraint* '}'
Group ::= 'group' Cardinality '{' ['opt'] FeatureDecl (',' ['opt'] FeatureDecl)*
'}'
Cardinality ::= 'allof' | 'oneof' | '[' IntLiteral '..' Limit ']'
AttributeDecl ::= 'Int' AName ';'
                | 'Int' AName in '[' Limit '..' Limit ']' ';'
                | 'Bool' AName ';'
                | 'String' AName ';'
Limit ::= IntLiteral | '*'
Constraint ::= Expr ';'
             | 'ifin'':'  Expr ';'
             | 'ifout'':' Expr ';'
             | 'require'':' FName ';'
             | 'exclude'':' FName ';'
Expr ::= 'True'
       | 'False'
       | IntLiteral
       | StringLiteral
       | FName
       | AName
       | FName '.'AName
       | UnOp Expr
       | Expr BinOp Expr
       | '(' Expr ')'
UnOp ::= '~' | '-'
BinOp ::= '||' | '&&' | '->' | '<->' | '=='
        | '!=' | '>'  | '<'  | '>='  | '<='
        | '+'  | '-'  | '*'  | '/'   | '%'
```

Attributes and values range over integers, strings or booleans.

The `FeatureDecl` clause specifies the details of a given feature, firstly by giving it a name (`FName`), followed by a number of possibly optional sub-features, the feature's attributes and any relevant constraints.

The `FeatureExtension` clause specifies additional constraints and attributes for a feature, and if the extended feature has no children a group can also be specified. This is particularly useful for specifying constraints that do not fit into the tree structure given by the root feature model.

Here is an example feature model for the `DeltaResourceExample` productline, defining valid combinations of features and valid ranges of parameters for cost, capacity and number of machines:

```
root Calculations {
  group oneof {
    Wordcount,
    Wordsearch
  }
}

root Resources {
  group oneof {
    NoCost,
    Cost { Int cost in [ 0 .. 10000 ] ; }
  }
}

root Deployments {
  group oneof {
    NoDeploymentScenario,
    UnlimitedMachines { Int capacity in [ 0 .. 10000 ] ; },
    LimitedMachines { Int capacity in [ 0 .. 10000 ] ;
      Int machinelimit in [ 0 .. 100 ] ; }
  }
}
```

ABS-Backends

16. Maude Backend

The Maude backend is a high-level, executable semantics in rewriting logic of the ABS language. Due to its relatively compact nature, it serves as a test-bed for new language features.

Executing a model on the Maude backend results in a complete snapshot of the system state after execution has finished.

The main drawback of the Maude backend is its relatively poor performance, making it not very suitable to simulate large models.

Features:

- CPU and bandwidth resources
- Simulation of resource usage on deployment components
- Timed semantics

- Executable formal semantics of the ABS language

16.1. How to run the Maude backend

Running a model on Maude involves compiling the code, then starting Maude with the resulting file as input, with the file `abs-interpreter.maude` accessible to Maude.

Compiling all files in the current directory into Maude is done with the following command:

```
$ absc -maude *.abs -o model.maude
```

The model is started with the following commands:

```
$ maude
Maude> in model.maude
Maude> frew start .
```

This sequence of commands starts Maude, then loads the compiled model and starts it. The resulting output is a dump of the complete system state after execution of the model finishes.

In case of problems, check the following:

- `absc` should be in the path; check the `PATH` environment variable.
- `absfrontend.jar` should be in the environment variable `CLASSPATH`.
- `abs-interpreter.maude` should be in the same directory as the compiled model, or in a directory listed in the environment variable `MAUDE_LIB`.

---

17. Erlang Backend

The Erlang backend runs ABS models on the Erlang virtual machine by translating them into Erlang and combining them with a small runtime library implementing key ABS concepts (cogs, futures, objects, method invocations) in Erlang.

Executing an ABS model in Erlang currently returns the value of the last

statement of the main block; output via `ABS.Meta.println` is printed on the console. More introspective and interactive capabilities are planned and will be implemented in the future.

17.1. How to run the Erlang backend

Running a model in Erlang involves compiling the ABS code, then compiling and running the resulting Erlang code.

Compiling all files in the current directory into Erlang is done with the following command:

```
$ absc -erlang *.abs
```

The model is started with the following commands, where `/Modulename/` should be the name of the module containing the main block:

```
$ erl
1> code:add_path("gen/erl/ebin").
2> cd ("gen/erl").
3> make:all([load]).
4> runtime:start("/Modulename/").
```

This sequence of commands starts Erlang, then compiles the generated Erlang code and starts it.

---

18. Haskell Backend

The Haskell backend translates ABS models to Haskell source code, consequently compiled through a Haskell compiler and executed by the machine. The backend, albeit a work in progress, already supports most ABS constructs and, above that, augments the language with extra features, such as `Type Inference`, `Foreign Imports` and real `Deployment Components`.

Type Inference

With the feature of `Type Inference` enabled, the user can *optionally* leave out the declaration of types of certain expressions; the backend will be responsible to infer those types and typecheck them in the ABS program. The type inference is *safe*, in the sense that it will not infer any wrong types (soundness property).

To make use of this feature, the user puts an underscore `_` in place of where a type would normally be, as in this ABS block of code:

```
{ _ x = 3;
  Int y = 4; // type inference is optional
  x = x+y;
  _ l = Cons(x, Cons(y, Nil));
  _ s = length(l) + 4; }
```

> At the moment, the type inference cannot infer *interface types* as in `_ o = new Class();`. It can however infer all the other types, that is Builtin, Algebraic, and Exception data types. There is a plan to support this in the future.

Foreign Imports

The Haskell backend extends the ABS module system with the ability to include Haskell-written code inside the ABS program itself. This feature is provided by the `foreign_import` keyword, which syntactically follows that of the normal `import` keyword. To illustrate this:

```
module Bar;
...
foreign_import Vertex from Data.Graph;
foreign_import vertices from Data.Graph;
```

the programmer has imported the `Vertex` algebraic datatype and the `vertices` function from the `Data.Graph` Haskell library module into an ABS module named `Bar`. Any imported Haskell term will be treated as its ABS counterpart. In the example case, the programmer may re-export the foreign terms or use them as normal ABS terms:

```
{
  Graph g = empty_graph();
  List<Vertex> vs = vertices(g);
}
```

At the moment, the ABS programmer can reuse (with `foreign_import`) Haskell's *Algebraic Data types* and *Pure functions*, but not monadic IO code (Haskell code with side-effects). This restriction is planned to be lifted in a later release of the backend.

Deployment Components

The Haskell backend implements the ABS feature of Deployment Components, faithfully as described in Chapter 8. The backend follows the view that Deployment Components are *virtual machines* running in the Cloud. As such, each single DC corresponds to one Cloud virtual machine (VM).

Two DC classes (implementations) are provided to support theOpenNebula and Microsoft Azure cloud computing platforms accordingly:

```
class NebulaDC(CPU cpu, Mem memory) implements DC {
  ...
}
```

```
class AzureDC(CPU cpu, Mem memory) implements DC {
  ...
}
```

The `CPU` and `Mem` datatypes are passed as arguments when creating the DC to parameterize its computing resources. These datatypes are simple defined as type synonyms to Int, but you can expect more sophisticated resource encodings for a future backend release.

```
type CPU = Int; // processor cores
type Mem = Int; // RAM measured in MB
```

The backend has only been developed on and tested against the OpenNebula platform. This hopefully will change when more cloud providers will be supported.

18.1. How to obtain and install

The compiler itself is written in Haskell and distributed as a normal Haskell package. Therefore to build abs2haskell you need either

1) a recent release of the Haskell platform (version >= 2013.2.0.0),

2) the GHC compiler accompanied by the Cabal packaging system:

- GHC compiler (version >=7.6)
- Cabal package (version >=1.4)
- `cabal-install` program. The compiler depends on other community packages/libraries. This program will automatically fetch and install any library dependencies.

Downloading, building and installing the compiler

Clone the repository with the command:

```
$ git clone git://github.com/bezirg/abs2haskell
```

To build and install the abs2haskell bakend run inside the `abs2haskell/` directory:

```
sudo make install
```

18.2. How to run the Haskell backend

After installing the compiler, you should have the program `abs2haskell` under your `PATH`.

Examples of running:

```
$ abs2haskell Example.abs

# An ABS program may have multiple main blocks in different modules.
# So you have to specify in which module is the main block you want to build with

$ abs2haskell --main-is=Example.abs Example.abs

$ abs2haskell examples/   # will compile all ABS files under examples directory
```

The compiler will generate ".hs" files for each compiled ABS module. No other runtime system libraries and dependencies will be generated.

The final step before running the ABS program is to compile the generated

Haskell code to machine code, as the example:

```
ghc --make -threaded Example.hs # put the generated haskell file that has the main
block here
```

Running the final program

```
./Example -O # means run it on 1 core with default optimizations
./Example -O +RTS -N1 # the same as the above
./Example -O +RTS -N2 # run it on 2 cores
./Example -O +RTS -N4 # run it on 4 cores
./Example -O +RTS -NK # run it on K cores
```