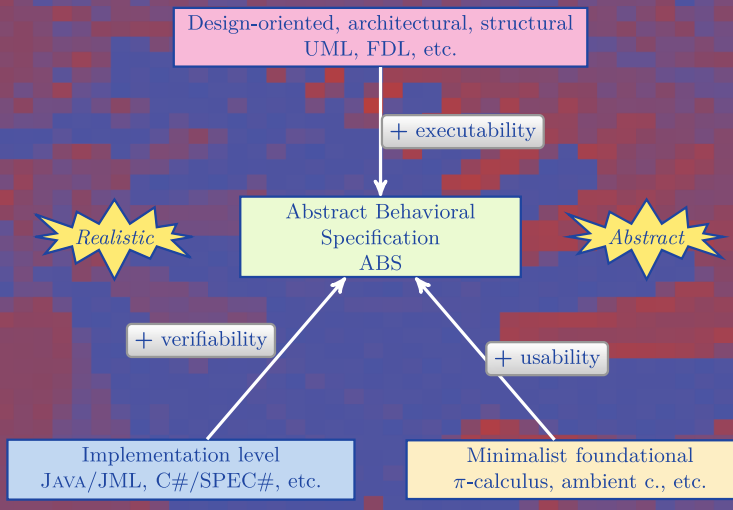
State-of-the-Art Survey

LNCS 7866

Elena Giachino
Reiner Hähnle
Frank S. de Boer
Marcello M. Bonsangue (Eds.)

Formal Methods for Components and Objects

11th International Symposium, FMCO 2012
Bertinoro, Italy, September 2012
Revised Lectures



Springer

Lecture Notes in Computer Science 7866

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

Elena Giachino Reiner Hähnle
Frank S. de Boer Marcello M. Bonsangue (Eds.)

Formal Methods for Components and Objects

11th International Symposium, FMCO 2012
Bertinoro, Italy, September 24-28, 2012
Revised Lectures

Springer

Volume Editors

Elena Giachino
University of Bologna, Dept. of Computer Science
Mura Anteo Zamboni, 7, 40127 Bologna, Italy
E-mail: giachino@cs.unibo.it

Reiner Hähnle
Technical University of Darmstadt, Dept. of Computer Science
Hochschulstr. 10, 64289 Darmstadt, Germany
E-mail: haehnle@cs.tu-darmstadt.de

Frank S. de Boer
Centre for Mathematics and Computer Science, CWI
Science Park 123, 1098 XG Amsterdam, The Netherlands
E-mail: f.s.de.boer@cwi.nl

Marcello M. Bonsangue
Leiden University, Leiden Institute of Advanced Computer Science (LIACS)
P.O. Box 9512, 2300 RA Leiden, The Netherlands
E-mail: marcello@liacs.nl

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-40614-0 e-ISBN 978-3-642-40615-7
DOI 10.1007/978-3-642-40615-7
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013946244

CR Subject Classification (1998): D.2.4, D.2, F.3, F.4, D.3, D.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Modern software systems are complex and often structured as a composition of a high number of components or objects. In order to construct such complex systems in a systematic manner, the focus in development methodologies is on structural issues: Both data and functions are encapsulated into software units that are integrated into large systems by means of various techniques supporting reusability and modifiability. This encapsulation principle is essential to both the object-oriented and the component-based software engineering paradigms.

Formal methods for component- and object-oriented systems are needed to ensure behavioral and security guarantees, with special emphasis on specification, modeling and validation techniques supporting the concepts of reusability, adaptability and evolvability of the systems, with which the systems can cope with changes in the environment as well as with modified and new requirements.

The 11th Symposium on Formal Methods for Components and Objects (FMCO 2012) was held during September 24–28, 2012, as an international school at Centro Residenziale Universitario (CRU) of the University of Bologna, located in Bertinoro, a small medieval hilltop town in Italy. FMCO 2012 was organized by the European project HATS (*Highly Adaptable and Trustworthy Software using Formal Models*), a European Integrated Project within the FET *Forever Yours* programme, in agreement with the EternalS Coordination Action (CA) that coordinates research among the four projects of the *Forever Yours* initiative: LivingKnowledge, HATS, Connect, and SecureChange.

FMCO 2012 featured lectures by world-renowned experts in the area of formal models for objects and components. This volume contains the revised papers submitted by the lecturers. The proceedings of the previous editions of FMCO have been published as volumes 2852, 3188, 3657, 4111, 4709, 5382, 5751, 6286, 6957, and 7542 of Springer's *Lecture Notes in Computer Science*. We believe that this volume and all previous proceedings provide a unique combination of ideas on software engineering and formal methods that reflect the expanding body of knowledge on modern software systems.

Finally, we thank all authors for the high quality of their contributions, and the reviewers for their help in improving the papers in this volume.

June 2013

Frank de Boer
Marcello Bonsangue
Elena Giachino
Reiner Hähnle

Organization

FMCO 2012 was organized by the University of Bologna, Italy, in close collaboration with the Technical University of Darmstadt, Germany, the Centrum voor Wiskunde en Informatica (CWI), Amsterdam, and Leiden University, The Netherlands.

Program Organizers

Einar Broch Johnsen	University of Oslo, Norway
Reiner Hähnle	Technical University of Darmstadt, Germany
Arnd Poetzsch-Heffter	Technical University of Kaiserslautern, Germany
German Puebla	Universidad Politecnica de Madrid, Spain
Davide Sangiorgi	University of Bologna, Italy

Local Organizers

Mario Bravetti	University of Bologna, Italy
Elena Giachino	University of Bologna, Italy
Davide Sangiorgi	University of Bologna, Italy

Sponsoring Institutions

European project HATS (FP7-231620)
European Coordination Action EternalS

Table of Contents

The Abstract Behavioral Specification Language: A Tutorial Introduction*

Reiner Hähnle

Department of Computer Science, Technische Universität Darmstadt
haehnle@cs.tu-darmstadt.de

Abstract. ABS (for abstract behavioral specification) is a novel language for modeling feature-rich, distributed, object-oriented systems at an abstract, yet precise level. ABS has a clear and simple concurrency model that permits synchronous as well as actor-style asynchronous communication. ABS abstracts away from specific datatype or I/O implementations, but is a fully executable language and has code generators for Java, Scala, and Maude. ABS goes beyond conventional programming languages in two important aspects: first, it embeds architectural concepts such as components or feature hierarchies and allows to connect features with their implementation in terms of product families. In contrast to standard OO languages, code reuse in ABS is feature-based instead of inheritance-based. Second, ABS has a formal semantics and has been designed with formal analyzability in mind. This paper gives a tutorial introduction to ABS. We discuss all important design features, explain why they are present and how they are intended to be used.

1 Introduction

Software used to be written for (i) a dedicated purpose to be (ii) deployed in a specific environment and (iii) to be executed on a stand-alone machine. This situation changed drastically: all consumer appliances of a certain complexity, from washing machines via mobile phones to vehicles, contain large amounts of software. High diversification and rapid pace of change dictated by contemporary market conditions require that this software is able to cope with an extreme degree of variability and adaptability. Planned reuse is not just an option, but a key strategy to staying competitive.

At the same time, modern software is nearly always concurrent and mostly also distributed. It is hard to imagine state-of-art business software that is not based on some notion of distributed services. A more recent trend is virtualization: as more and more software is deployed in the cloud, one consequence is that clients loose to some extent control over the execution environment: the exact architecture, the number of processors, the load, as well as other deployment parameters are typically not available at the time when software is developed.

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

E. Giachino et al. (Eds.): FMCO 2012, LNCS 7866, pp. 1–37, 2013.

Because of this, the trend to virtualization leads to a new potential gap in the software development chain between developers and operators.

In a software development scenario where one has to deal with extreme variability, with complex distributed computation, and with the need to abstract from deployment issues, the availability of suitable *software modeling languages*, as well as powerful tools helping in *automation*, becomes crucial.

Design-oriented and architectural languages, notably the UML family of notations, cannot fulfill this role, because they lack executability and mathematical rigor. Executable formalisms for specifying concurrent behavior, such as state charts [28], process calculi [38], abstract state machines [7], or Petri nets [24], are simply too minimalist to describe industrial systems. In addition, they are not integrated with architectural notations or with feature description languages [45] that model variability. The latter, however, do not provide a connection between features and their realization. Refinement-based approaches, such as Event-B [1] require too much rigor in their application for being feasible outside extremely safety-critical applications. They also do not address variability. Finally, implementation-oriented specification languages, such as JML [34] (for JAVA) or SPEC# [6] (for C#) inherit all the complications and idiosyncrasies of their host languages and are not very good at specifying concurrent behavior.

Our brief analysis exhibits a *gap* in the landscape of software specification and modeling languages. The European FP7 Integrated Project HATS (*Highly Adaptable & Trustworthy Software Using Formal Models*) developed the *Abstract Behavioral Specification* (ABS) language in order to address this issue. ABS is a software modeling language that is situated between architectural, design-oriented, foundational, and implementation-oriented languages, see Fig. 1.

1.1 Structure of This Chapter

In this chapter we give a tutorial introduction into the design principles, language elements, and usage of the ABS language. We discuss the design considerations behind ABS in Sect. 2 and give an architectural overview in Sect. 3. Then we present different language layers, starting with the functional layer in Sect. 4, followed by the OO-imperative layer in Sect. 5, the concurrency layers in Sect. 6 and language extensions based on pluggable type systems as well as a foreign language interface (Sect. 7). On top of these layers are language concepts for modeling of software product lines. These are discussed in Sect. 8. We close the tutorial with some general considerations on modeling and a discussion of current limitations of ABS in Sect. 9.

1.2 Further Reading

This paper is a tutorial on ABS and not a language specification nor a formal definition. A more technical and more detailed description of ABS and its tool set is contained in the paper trio [9,25,31]. The most detailed document about ABS that also contains a formal semantics is [17]. The official ABS Language Specification is [2]. The main web resources for ABS are `http://www.hats-project.eu`

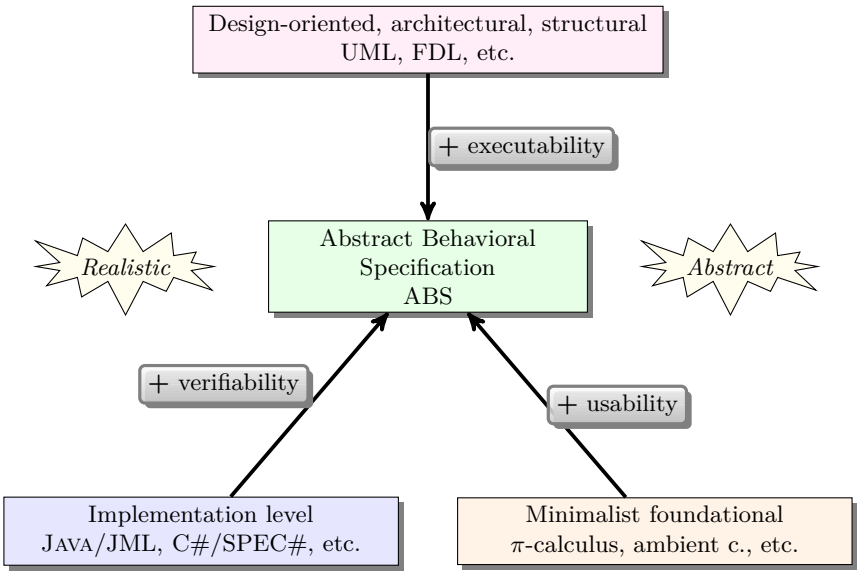


Fig. 1. The gap in the landscape of software modeling languages

and `www.abs-models.org`. Finally, for several case studies done with ABS, one can have a look at the public HATS Deliverable D5.3 [19].

It is stressed at several places in this tutorial that ABS has been designed with the goal of permitting automatic static analyses of various kinds. This tutorial concentrates on the ABS language and its development environment. In the paper by Albert et al. in this volume [4] automated resource analysis for ABS is explained in detail. Information on deadlock analysis and formal verification of ABS can be found in [18]. The chapter by Kurnia & Poetzsch-Heffter in this volume [33] contains a general discussion of verification of concurrent open systems such as ABS models.

1.3 Installation of the ABS Eclipse Plugin

For trying out the examples provided in this tutorial you will need the ABS Eclipse plugin. To install it, follow the simple instructions at `http://tools.hats-project.eu/eclipseplugin/installation.html`. You will need at least Eclipse version 3.6.2 and it is recommended to work with a clean installation.

The example project used throughout this tutorial is available as an archive from `http://www.hats-project.eu/sites/default/files/TutorialExample.zip`. To install, unzip the archive file into a directory `/mypath/Account`. Then create a new *ABS Project* in Eclipse and import the directory file contents into the workspace in the usual way. This opens automatically the ABS Modeling Perspective. After opening a few files in the editor you should see a screen similar to the one in Fig. 2.

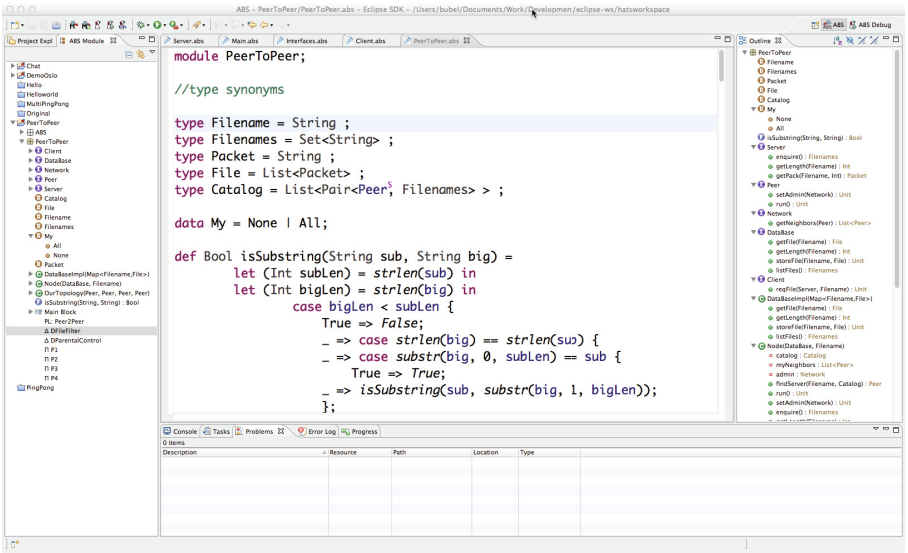


Fig. 2. Screenshot of ABS Modeling Perspective in ECLIPSE plugin

2 Design Principles of ABS

ABS targets software systems that are concurrent, distributed, object-oriented, built from components, and highly reusable. To achieve the latter, we follow the arguably most successful software reuse methodology in practice: *software product families* or *software product lines* [41], see also the Product Line Hall of Fame at `http://splc.net/fame.html`. Going beyond standard OO concepts to support the modeling of variability, ABS integrates feature models as a *first-class language concept*. As shown in Sect. 8, ABS thus provides language-based support for *product line engineering* (PLE).

As an abstract language ABS is well suited to model software that is supposed to be deployed in a virtualized environment. To close the gap between design and deployment it is necessary to represent low-level concepts such as system time, memory, latency, or scheduling at the level of abstract models. In ABS this is possible via a flexible and pluggable notation called *deployment components*. This goes beyond the present, introductory tutorial, but is covered in detail in the chapter by Johnsen in this volume [30].

ABS is not merely a modeling notation, but it arrives with an integrated tool set that helps to *automate the software engineering process*. Tools are useless, however, unless they ensure *predictability of results*, *interoperability*, and *usability*. A fundamental requirement for the first two criteria is a uniform, formal semantics. But interoperability also involves the capability to connect with other notations than ABS. This is ensured by providing numerous language

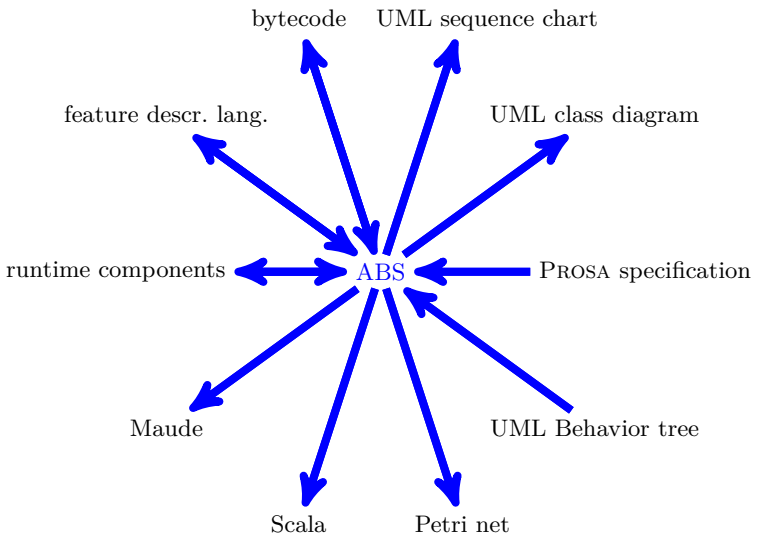


Fig. 3. Some interfaces between ABS and other languages

interfaces from and to ABS as shown in Fig. 3. These are realized by various import, export, and code generation tools, several of which are discussed below.

Arguably the most important criterion for tools, however, is usability. This tutorial is not the place to embark on a full discussion of what that entails, but it should be indisputable that *automation*, *scalability*, and *integration* are of the utmost importance.

To ensure the first two of these qualities, the HATS project adopted as a central principle to develop ABS *in tandem* with its tool set. This is not merely a historical footnote, but central to an understanding of the trade-offs made in the design of the ABS language. For most specification and programming languages their (automatic) analyzability is considered in hindsight and turns out not to be scalable or even feasible. With ABS, the slogan of *design for verifiability* that originated in the context of hardware description languages [37], has been systematically applied to a software modeling language. For example, the concurrency model of ABS is designed such that it permits a compositional proof system [3,21], the reuse principle employed in ABS is chosen in such a way that incremental verification is possible [26,27], etc.

Many formal methods tools focus on analysis, in particular, on verification. Functional verification, model checking, test case generation, and resource estimation are supported by ABS tools as well. Just as important as analytic methods, specifically in a model-based context, are *generative* ones: ABS is fully executable (albeit in a non-deterministic manner) and supports code generation to JAVA, SCALA, and MAUDE. In addition, it is possible to *learn* ABS models from observed behavior [20].

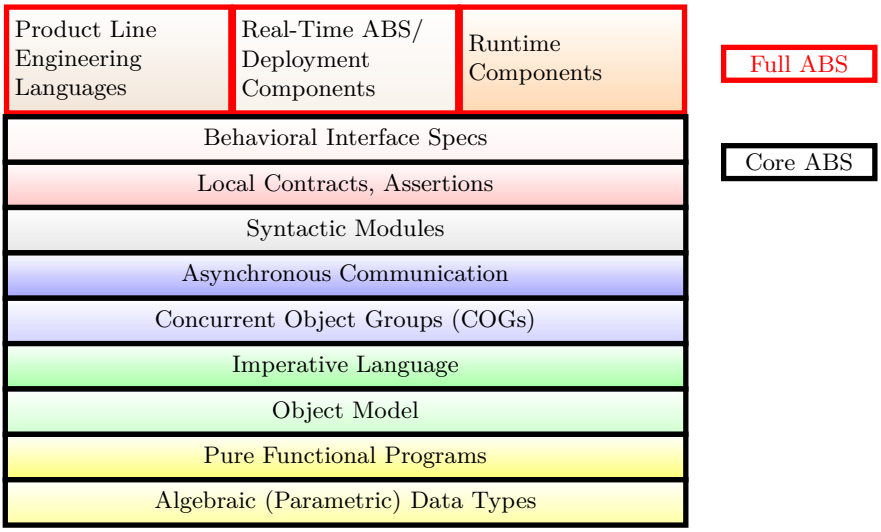


Fig. 4. Architecture of the ABS language

Regarding *integration*, the tool set around the ABS language is realized as a set of plugins for the popular Eclipse IDE. These plugins realize the *ABS Modeling Perspective* (see Fig. 2) and the *ABS Debug Perspective* (see Fig. 8), which provide about the same functionality as their Java counterparts, that is, parsing, syntax highlighting, parse error location, symbol lookup, compilation, building, runtime configurations, interactive debugging, etc. In addition to these standard development tools, however, a number of analysis and generative tools are available. Some of these, for example, Java code generation or type inference are illustrated below. An overview of the ABS tool suite is given in [48].

3 Architecture of ABS

The architecture of ABS has been organized as a stack of clearly separated layers as illustrated in Fig. 4. In the design of ABS we strove for

1. an attractive, easy-to-learn language with a syntax that is familiar to many developers and
2. a maximal separation of concerns (orthogonality) among different concepts.

The four bottom layers provide a modern programming language based on a combination of algebraic data types (ADTs), pure functions, and a simple imperative-OO language. The idea is that anyone familiar with the basic concepts of OO-imperative and functional programming as in the programming

languages Java and Haskell or Scala, is able to grasp this part of ABS immediately, even though ABS is considerably simpler than any of these languages.

The next two layers realize tightly coupled and distributed concurrency, respectively. The concurrency and synchronization constructs are designed in a way to permit a compositional proof theory for functional verification in a program logic [3, 21]. Standard contracts are used for functional specification of sequential programs and behavioral interfaces over sets of histories are used for specifying concurrent programs, see also the paper by Kurnia & Poetzsch-Heffter in this volume [33].

The language layers up to here are often called *Core ABS*. Above these are orthogonal extensions for product line engineering, deployment components, and runtime components that allow to model mobile code. The latter are not discussed in this volume, but are described in [35].

As mentioned above, ABS is a fully executable language. Nevertheless, *abstraction* is achieved in a number of ways: first of all, ABS contains only five built-in datatypes—everything else is user-defined. The rationale is that no premature decision on the properties of datatypes is enforced, which helps to create implementation-independent models. Second, functions on datatypes can be *underspecified*. The modeler has the alternative to return abstract values or to leave case distinctions incomplete. The latter may result in runtime errors, but is useful for simulation, test generation or verification scenarios. Third, the scheduling of concurrent tasks as well as the order of queuing messages is nondeterministic.

Of course, one might want to expose full implementation details of an abstract description at some time. This is possible by refining an ADT into an implemented class or by realizing it in Java via the *foreign language interface* available in ABS (Sect. 7.2). Concrete schedulers can be specified via deployment components [30, 39].

Crucially, the abstraction capabilities of ABS allow to specify partial behavior during early design stages, such as feature analysis, without committing to implementation details. This lends support, for example, to rapid prototyping or to the early analysis of the consequences of design decisions.

ABS has been designed as a compact language. Each layer has no more first-class concepts than are needed to ensure usability (including some syntactic sugar). This does not mean, however, that ABS is a small language: the ABS grammar has considerably more non-terminals than that of Java! The reason is that ABS features several language concepts that are simply not present in Java. This reflects the 7ambition of ABS to cover the whole modeling spectrum from feature analysis, deployment mapping, high-level design and down to implementation issues. To show that in spite of this ABS is not unwieldy, but rather easy to use and quite flexible, is the aim of the present tutorial.

4 The Functional Layer

4.1 Algebraic Data Types

The base layer of ABS is a simple language for parametric algebraic data types (ADTs) with a self-explaining syntax. The only predefined datatypes[1] are `Bool`, `Int`, `String`, `Unit` and parametric lists as defined below. The type `Unit` is used as a type for methods without return value and works like Java's `void`. All other types are user-defined. There is no subtyping and type matching is defined in the usual manner, see [15, Sect. 4.2] for details.

To be able to follow and try out the examples in this tutorial, it is strongly recommended that the Eclipse ABS plugin is installed (see Sect. 1.3). To create a new ABS file in an existing ABS project, right click on the project in the explorer and choose `New|ABS Module`. In the input mask that pops up, specify a file name, for example, `CustomerData`, and click `Finish`. This creates a new file named `CustomerData.abs` in the workspace, which can be edited with the Eclipse editor in the usual manner. The running example of this tutorial has a banking theme. Let us create some datatypes related to customers. Please note that *all type and constructor names must be upper-case.*

```
data Level = Standard | Silver | Gold;
data Customer = Test | Person(Int, Level) | Company(Int);
```

There are three kinds of customers defined by three different type constructors: a test customer, individual customers, and corporate customers. Individual customers are identified by a personal id and possess a status level while corporate customers are identified by their tax number. We can make the intention more explicit and, at the same, automatically create selector functions for the arguments of each type constructor by the following alternative definition:

```
data Customer = Test | Person(Int pid, level Level) |
                Company(Int taxId);
Customer p = Person(17, Silver);
Int n = pid(p);
```

Assume now that we want to define lists of customers. For this we can use the following built-in parametric list type, which provides a convenient concrete syntax for lists:

```
data List<T> = Nil | Cons(T, List<T>);
List<Int> l = [1,2,3];
```

The definition of parametric lists demonstrates that type definitions may be recursive. Let us instantiate parametric lists with `Customer` and, at the same time, create a *type synonym*:

[1] There is one more predefined type that is used for synchronization which is explained in Sect. 6.3.

```
type CustomerList = List<Customer>;
```

Type synonyms do not add new types or functionality, but can greatly enhance readability.

4.2 Functions

The functional layer of ABS consists of a pure first-order functional language with definitions by case distinction and pattern matching. *All function names must be lower-case.* Let us begin by defining a function that computes the length of a list of customers:

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

Several kinds of patterns are permitted on the left-hand side of case distinctions. In the example above, the first and second cases use a data constructor pattern. In the second case, this contains an unbound variable whose value is extracted and used on the right-hand side. The last case uses an underscore pattern containing an anonymous variable that matches anything. As long as `CustomerList` is defined as above, the last case is never executed, however, one could justify its presence as a defensive measure in case someone adds additional list constructors.

Naturally, it would have been possible to define a parametric version of the function `Int length(List<T> list)`. This is left as an exercise to the reader.

Here is another example illustrating the remaining patterns, that is, the literal pattern and the bound variable pattern:

```
def Int sign(Int n) =
  case n {
    0 => 0 ;
    n => if (n > 0) then 1 else -1 ;
  } ;
```

The ABS parser does not attempt to establish whether case distinctions are exhaustive. If no pattern in a case expression matches, a runtime error results. It is up to the modeler to prevent this situation (in the near future, ABS will be equipped with the possibility of failure handling). Similarly, it is perfectly possible to define the following function:

```
def Int getPid(Customer c) = pid(c);
```

However, if `c` has any other type constructor than `Person(Int,Level)` at runtime, an error will result.

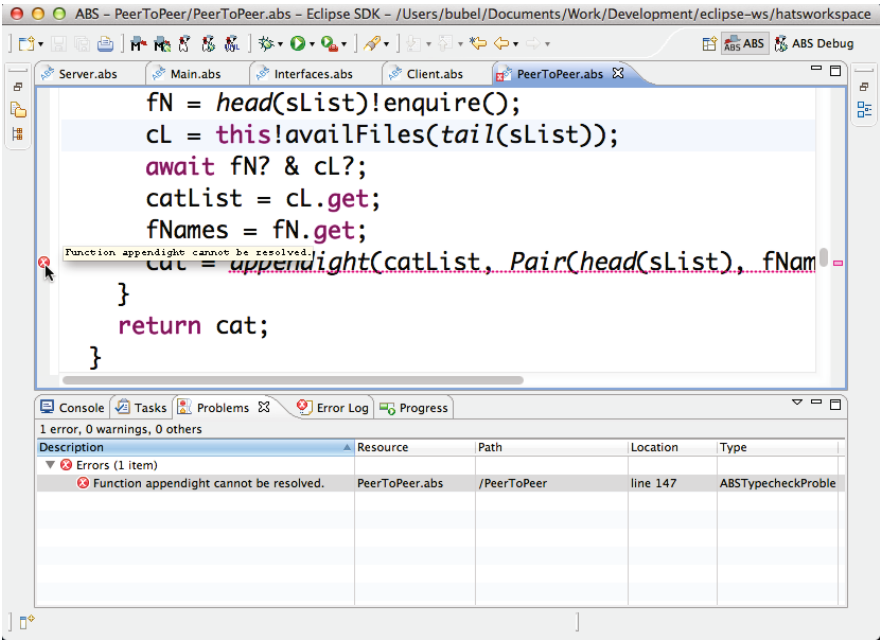


Fig. 5. Parse error in the ABS Eclipse editor; errors are highlighted on the editor line where they occur as well as in the Problems tab of the messages subwindow

We close this section by illustrating a piece of syntactic sugar for associative collection types such as sets, bags, sequences, etc. To construct concrete elements of such datatypes one typically needs to iterate several binary constructors, such as `Insert(1,Insert(2,Insert(3,EmptySet)))`. This is cumbersome. The following idiom defines an n-ary constructor that uses concrete list syntax as its argument. By convention, the constructor should have the same name as the type it is derived from, but in lower-case.

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

4.3 Modules

If you tried to type in the previous examples into the ABS Eclipse editor you got parse errors despite the definitions being syntactically correct (similarly as in Fig. 5). This is, because any ABS definition must be contained in exactly one

module. ABS is equipped with a simple syntactic module system that is inspired by that of HASKELL [40]. To make the examples of the previous section work, simply add a *module declaration* like the following as the first line of the file:

```
module CustomerData;
```

Module names must be upper-case and define a syntactic scope until the end of the file or until the next module declaration, whichever comes first. Module names can also be part of qualified type names.

Module declarations are followed by *export* and *import* directives. The former lists the types, type constructors, and functions that are visible to other modules, the latter lists the entities from other modules that can be used in the current module. With the type definitions of the previous section we might write:

```
module CustomerData;
export Standard, Customer, Company, getPid;
...

module Test;
import * from CustomerData;
def Customer c1() = Company(2);
def Customer c2() = Person(1,Standard); // erroneous
```

The module `CustomerData` exposes three of its constructors and a function while module `Test` imports anything made available by the former. The definition of `c1` is correct, but the definition of `c2` gives a parse error about a constructor that cannot be resolved, because `Person` is not exported.

The `from` clause constitutes an *unqualified import*. Instead, it is also possible to make *qualified* imports. For example, we could have written:

```
module Test;
import CustomerData.Company, CustomerData.Customer;
def CustomerData.Customer c1() = CustomerData.Company(2);
```

In this case, however, one must also use qualified type names in the definitions as illustrated above, which can easily become cumbersome and hard to read.

The ABS compiler knows one predefined module that does not need to be explicitly imported—the ABS standard library `ABS.StdLib`. It contains a number of standard collection types, such as lists, sets, maps, together with the usual functions defined on them. It also contains some other types and functions that are used often. The standard library module is contained in a file named `abslang.abs`. Specifically, it contains definitions of the parametric datatypes `Set<T>` and `List<T>` discussed above, together with constructors and standard functions such as `add`, `length`, etc. Therefore, it is not necessary to define these types in `CustomerData.abs`.

To look up the definition of any standard type or function (or any other type or function, for that matter), simply move the cursor over the identifier in question and press F3. For example, pressing F3 over the identifier `Cons` in

the definition of `length` in the previous section opens a new tab in the ECLIPSE editor that contains `abslang.abs` and jumps to the line with the definition of the constructor `Cons`. This lookup functionality is, of course, well-known to users of the ECLIPSE IDE.

4.4 Abstract Data Types

The module system allows to define *abstract data types* by hiding the type constructors. This implies that only functions can be used to access data elements. Their explicit representation is hidden. Of course, one then needs to supply suitable constructor functions, otherwise, no new elements can be created at all.

In the example of the previous section we might decide to expose only the types and constructor functions as follows:

```
module CustomerData;
export Customer, Level, createCustomer, createLevel;
def Customer createCustomer(Int id, String kind) = ... ;
def Level createLevel(String kind) = ... ;
```

We leave it as an exercise for the reader to write a suitable definition of `createCustomer(Int,String)` and `createLevel(String)`. As usual, the advantage of using abstract data types is that one can replace the definition of types without the need to change any client code.

5 The OO-Imperative Layer

5.1 The Object Model

ABS follows a strict *programming to interfaces* discipline [22]. This means that the *only* declaration types of objects are interface types. Consequently, ABS classes do *not* give rise to type names.

Apart from that, interface declarations are pretty standard and follow a JAVA-like syntax. They simply consist of a number of method signatures. Static fields are not permitted,[2] but subinterfaces, even multiple subinterfaces, are allowed. Let us give an example that models the Customer type from previous sections in an object-oriented fashion. The file `Customer.abs` contains the following definitions:

```
module CustomerIF;

export Customer;

import Level, createLevel from CustomerData;

interface Customer { Int getId(); }
interface IndvCustomer extends Customer {}
interface CorpCustomer extends Customer { Level getLevel(); }
```

[2] And neither are static classes and objects. Instead of static elements the ABS modeler should consider to use ADTs and functions.

As can be seen, interfaces and classes can be exported. In fact, this is necessary, if anyone outside their defining module should be able to use them.

It is possible to mix object and data types: data types may occur anywhere within classes and interfaces as long as they are well-typed. Less obviously, reference types may also occur inside algebraic data types. As seen earlier, it is perfectly possible to declare the following type:

```
type CustomerList = List<Customer>;
```

Keep in mind, though, that it is not allowed to call methods in function definitions. The reason is that method calls might have side effects.

It was mentioned already that classes do not provide type names. They are only used for object construction. As a consequence, in ABS it is always possible to decide when a class and when an interface name is expected. Therefore, interfaces and classes may have the same name. We do not recommend this, however, because it dilutes the programming to interfaces discipline. It is suggested to use a base name for the interface and derive class names by appending "Impl" or similar.

A class may implement multiple interfaces. Class constructors are not declared explicitly, instead, class declarations are equipped with parameter declarations that implicitly define corresponding fields and a constructor. Class definitions then consist of field declarations, followed by an initialization block and method implementations. Any of these elements may be missing. Hence, we can continue the example as follows:

```
class CorpIndvCustomerImpl(Int id)
  implements IndvCustomer, CorpCustomer {
  Level lv = createLevel("Standard");
  // no initialization block present
  Level getLevel() { return lv; }
  Int getId() { return id; }
}
```

Here, the `id` field is modeled as a class parameter, because it is not possible to give a reasonable default value, which would be required for an explicit field declaration: fields that have no reference type must be initialized. Reference type fields are initialized with **null**.

In contrast to functions, method names need not (and cannot) be exported by modules. It is sufficient to get hold of an object in order to obtain access to the methods that are defined in its class.

The most striking difference between ABS and mainstream OO languages is that in ABS there is *no class inheritance* and, therefore, also *no code inheritance*. So, how do we achieve code reuse? In ABS we decided to disentangle data design and functionality from the modeling of code variability. For the former, we use functions and objects (without code inheritance), whereas for the latter we use a layer on top of the core ABS language that permits to connect directly with feature models. This layer is discussed in Sect. 8.

In a concurrent setting (see Sect. 6) one typically wants some objects to start their activity immediately after initialization. To achieve this in ABS, one can define a `Unit run()` method, which implicitly declares a class as *active* and lets objects execute the code in the body of the run method after initialization. Classes without a `run()` method are called *passive* and their objects react only to incoming calls.

5.2 The Imperative Layer

Statements. ABS has standard statements for sequential composition, assignment, while-loops, conditionals, synchronous method calls, and method return. As an example for all of these, look at the following implementation of the method `findCustomer(CustomerList)` in class `CorpIndvCustomerImpl` (don't forget to add it to the implemented interfaces as well to render it visible!).

```
Customer findCustomer(CustomerList cl) {
  Customer result;
  Int i = 0;
  while (i<length(cl)) {
    Customer curCust = nth(cl,i);
    Int curId = curCust.getId();
    if (id==curId) {result = curCust;}
    i = i + 1;
  }
  return result;
}
```

In addition to the various constructs, we can illustrate several ABS-specific restrictions with this example: first, it is necessary that the final statement in the method body is a return statement with the correct type. A typical ABS idiom is, therefore, to declare a local `result` variable. Neither for-loops nor breaks from loops are supported at the moment. To avoid going through the remaining list after the element has been found (as done here), one would need to add a `Bool found` variable and test that in the loop guard. Complex expressions are not allowed at the moment in tests of conditionals or loops. The workaround, as shown above, is to declare a local variable `curId` that holds an intermediate result. While these restrictions can be slightly annoying they hardly matter very much. It is likely that some of them will be lifted in the future, once it is better known what modelers wish.

Object Access. A more fundamental restriction concerns the usage of fields in assignment statements: assignments to fields and field lookups are only possible for the current object. Given a field `f`, an assignment of the form "`x = f;`" is always implicitly qualified with the current object as in "`x = this.f;`"; likewise an assignment to a field of the form "`f = exp;`" is always implicitly qualified as in "`this.f = exp;`". This implies that fields in ABS are *object private*. They cannot be directly seen or be modified by other objects, not even by objects from the

same class (as is possible even for private fields in JAVA). In other words, ABS enforces *strong encapsulation* of objects: it is only possible to view or change the state of another object via getter- and setter-methods. For example, it is not possible to change the second line in the body of the while-loop of `findCustomer(CustomerList)` as follows:

```
Int curId = curCust.id;
```

The designers of ABS consider enforcement of object encapsulation not as a restriction, but as a virtue: it makes all cross references between objects syntactically explicit. This avoids errors that can be hard to find. In addition it makes static analysis much more efficient, because any cross reference and possible side effect to a different object can be associated with a unique method contract.

For the practicing ABS modeler object encapsulation is greatly alleviated by the method completion feature of the ECLIPSE editor: if one types the beginning of the example above "`Int curId = curCust.`", then a pop-up menu will offer a selection of all methods that are known for objects of type `Customer`, the getter-method `getId()` among them. If a suitable method is not found, then the modeler can take this as a hint that it needs to be implemented or added to an interface.

Blocks. ABS is a *block-structured* language. Blocks are delimited by curly braces (no semicolon afterward) and may appear at four different places in a program:

1. as a way to group several statements and provide a scope for local variables—blocks are necessary for bodies of loops and conditionals that have more than one statement;
2. as method bodies;
3. as the (optional) class initialization block, between field and method declarations;
4. as an (optional) implicit "main" method at end of a module.

The last usage serves as an entry point for execution of ABS programs. At least one main block in one module is necessary for executing an ABS project, otherwise it is not clear which objects are to be created and executed. Any module that has a main block is selectable as an execution target via the ECLIPSE Run Configurations ... dialog or simply by right clicking on the desired module in the explorer and selection of Run As. We might complete our example now by specifying the following main block for module `CustomerIF`:

```
{
  Customer c = new CorpIndvCustomerImpl(17); // create some customers
  Customer d = new CorpIndvCustomerImpl(16);
  CustomerList l = Cons(c,Cons(d,Nil));      // create list of customers
  Customer e = c.findCustomer(l);            // we should find c in l
}
```

This code illustrates at the same time the usage of the **new** statement, which works as in JAVA. As usual in ECLIPSE, pressing the F4 key displays the type hierarchy pertaining to a class or interface name at the cursor position.

6 The Concurrency Layers

6.1 Background

One of the most distinctive features of ABS is its concurrency model. If we look at commercial programming languages such as C, C++, or Java, one can observe that, despite intense efforts in the last decade, none of them has a fully formalized concurrency model. Even though there are promising efforts towards a formal concurrency model of Java [5], the details are so complex that they are likely to compromise usability of any resulting system. The reason is that current industrial programming languages have a very low-level concurrency model and do not natively support distributed computation. This has practical consequences, such as burdening the programmer with, for example, taking care of prevention of data races.

A more fundamental problem is the difficulty to design a compositional proof system for such languages. By *compositionality* we mean that one can specify and verify the behavior of a single method in isolation from the rest of the system. This is a prerequisite for being able to deduce global behavior from the composition of local behavior. In a setting, where concurrent objects can arbitrarily cross-reference each other, this is hardly possible. Arbitrarily complex, global invariants, might be needed to describe behavior. One approach to tackle the problem is to impose structure on concurrent objects and to make their dependencies syntactically explicit. In the realm of Java, JCoBox [44] is a suitable framework. It has been simplified and renamed into *Concurrent Object Group* (COG) in the context of ABS. COGs constitute the *lower tier* of the ABS concurrency model and are intended for closely cooperating concurrent tasks.

A second shortcoming of mainstream programming languages is the lack of support for distributed computation, that is, asynchronous communication among nodes that do not share memory. This form of concurrency has been abstracted into the *Actor* model [29] and is realized with first-class support in recent languages such as Scala. ABS implements a version of Actor-based distributed computation where COGs form the primitive units of distribution. This constitutes the *upper tier* of the ABS concurrency model. Its main ideas are derived from the modeling language Creol [32].

6.2 Component Object Groups

An ABS Concurrent Object Group (COG) is a collection of tasks with shared memory and processor. This means that exactly one task is active at any given time and tasks can cross-reference each others' data. The situation can be visualized as in Fig. 6.

Within a COG, synchronous as well as asynchronous method calls are permitted. For the former, we use the standard syntax `target.method(arg1,arg2,...)`. *Synchronous method calls* within COGs represent sequential execution of code, that is, they block the caller and execute the code of the target until control is returned.

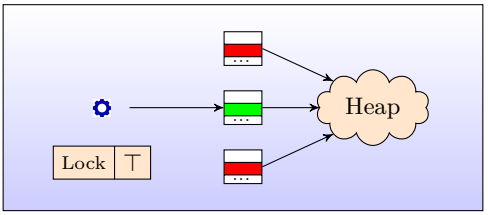


Fig. 6. Illustration of an ABS Component Group

Asynchronous method calls use the syntax `target!method(arg1,arg2,...)` and they cause the creation of a new task that is to execute the code of the target. Unlike in synchronous calls, execution of the code of the caller continues.

The main point to understand about COGs is that multitasking is not *preemptive* (decided by a scheduler). Rather it is an explicit decision of the ABS modeler when control is transferred to another task. To this end, ABS provides scheduling statements that allow *cooperative multitasking*. In between the explicit scheduling points, only one task is active, signified by the (single) *lock* of a COG being set to ⊤. As a consequence, data races between synchronization points simply cannot happen, which was an important design goal of ABS.

6.3 Scheduling and Synchronization

So, how are scheduling points specified in ABS? It is here that we encounter a second, central concurrency principle of ABS: *communication and synchronization are decoupled*. This is done via *future types* [13]. For any ABS type `T` a legal type name is `Fut<T>` and one can assign to it the result of any asynchronous method call with return type `T`. A variable declared with type `Fut<T>` serves as a reference to the future result of an asynchronous call and allows to retrieve it once it will have been computed. For example, the final line of the example on p. 15 can be rewritten into:

```
Fut<Customer> e = c!findCustomer(1);
<do something else>
```

Now the call creates a new task in the current COG and declares `e` as a future reference to its final result. The following code is executed immediately.

The future mechanism allows to dispatch asynchronous calls, continue with the execution, and then synchronize on the result, whenever it is needed. Synchronization is achieved by the command `await g`, where `g` is a polling *guard*. A guard is a conjunction of either side-effect free boolean expressions or *future guards* of the form `f?`. In the latter, `f` is a variable that has a future type. If the result to which `f?` is a reference is ready and available, then the expression evaluates to true. When the guard of an await statement evaluates to true, the computation simply continues. If, however, a guard is not true, then the current task releases the lock of its COG and gives another task in that COG the chance

to continue. When later the task is scheduled again, the guard is re-evaluated, and so on, until it finally becomes true. We call this a *conditional scheduling point* or *conditional release point*. To continue the previous example we could write:

```
await e?;
```

If the asynchronous call to `findCustomer(l)` has finished at this point, then execution simply continues. Otherwise, the lock of the current COG is set to $\perp$ and the processor is free to proceed with another task. For efficiency reasons ABS allows only *monotonic* guards and only conjunctive composition.[3] Once the result from an asynchronous call is available, it can be retrieved with a `get`-expression that has a future variable as its argument. In the example this may look as follows:

```
Customer f = e.get;
```

In summary, the following programming idiom for asynchronous calls and retrieving their results is common in ABS:

```
Fut<T> v = o!m(e); ... ; await v?; T r = v.get;
```

ABS does not attempt to check that each `get` expression is guarded by an `await` statement. So what happens when the result of an asynchronous call is not ready when `get` is executed? The answer is that the execution of the whole COG containing the task *blocks*. The difference between suspension and blocking is that in the latter case *no* task in same COG can continue until the blocking future is resolved.

Sometimes it is convenient to create an *unconditional scheduling point*, for example, to insert release points into long running sequential tasks. The syntax for unconditional scheduling statements in ABS is "`suspend;`".

6.4 Object and COG Creation

In the previous section we discussed the fundamental concurrency model of ABS, which is based on COGs. Whenever we create an object with the **new** statement, it is by default created in the same COG as the current task (see upper part of Fig. 7). This is not adequate for modeling *distributed computing*, where each node has its own computing resources (processor) and nodes are loosely coupled.

In an ABS model of a distributed scenario we associate one COG with each node. New COGs are implicitly created when specifying the `cog` keyword at object creation (see lower part of Fig. 7): this creates a new COG and places the new object inside it. At the moment, COGs are not first-class objects in ABS and are accessible only implicitly through their objects.[4] As a consequence, it is

[3] It is possible to support arbitrary boolean combinations over future guards, however, at the price of a much less efficient realization, see [15, pp. 33f] for technical details.

[4] There is an extension for ABS runtime objects that allows explicit and dynamic grouping of COGs [35].

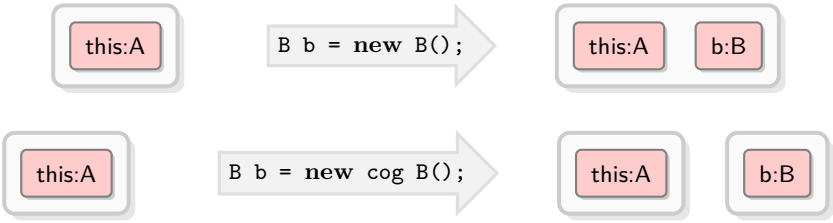


Fig. 7. Illustration of object and COG creation (gray boundaries represent COGs)

not possible to re-enter via recursive calls into the same execution thread. This is the reason why a simple binary lock for each COG is sufficient. Let us extend our running example with an `Account` class and COG creation in file `Account.abs`:

```
module Account;

interface Account {
        Int getAid();
        Int deposit(Int x);
        Int withdraw(Int x);
}

class AccountImpl(Int aid, Int balance, Customer owner)
      implements Account { ... }

{
  [Near] Customer c = new CorpIndvCustomerImpl(3);
  [Far] Account a = new cog AccountImpl(1,0,c);

  Fut<Int> dep = a!deposit(17);
  Fut<Int> wit = a!withdraw(17);

  await dep? & wit?;

  Int x = dep.get;
  Int y = wit.get;
  Int net = x + y;
}
```

In the main method of the `Account` module an account object `a` is created in a different COG from the current one. Note that there is no sharing of objects between COGs, so that the variable `c` provides no alias to the object parameter `c` in the constructor call `AccountImpl(1,0,c)`. The tasks resulting from the following two asynchronous calls will be executed on the node where `a` resides, which is different from the current one. A conjunctive guard ensures that the retrieval of the results is safe.

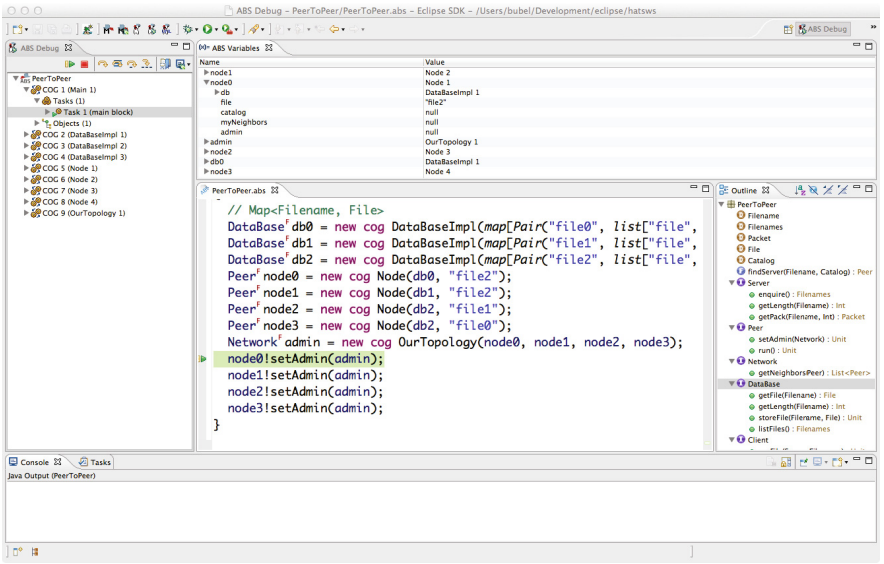


Fig. 8. Screenshot of ABS Debug Perspective in Eclipse plugin

It is possible to *animate* the execution of ABS code in two ways. To start the graphical ABS Debugger, simply right click on the file with the `Account` module in the explorer and select Run As|ABS Java Backend (Debug). This will automatically switch to the ABS Debug Perspective (see Fig. 8) and start the Eclipse debugger. All the usual features of a graphical debugger are available: navigation, breakpoints, state inspection, etc.

If instead, the backend Run As|ABS Java Backend (Debug with Sequence Diagram) is chosen, then in addition a UML sequence diagram that has a lifeline for each created COG is created and dynamically updated after each execution step in the debugger, see Fig. 9.

Location Types. Synchronous method calls to targets not in the current COG make no sense and are forbidden. For example, if we replace one of the asynchronous calls above with `a.deposit(17)`, a runtime error results. One possibility to avoid this is to *annotate* declarations with one of the types `Near` or `Far`, as shown above. This tells the compiler that, for example, `a` is in a different COG and cannot be the target of a synchronous call. Obviously, it is tedious to annotate all declarations; moreover, the annotations tend to clutter the models. To address this problem, the ABS Eclipse plugin implements a far/near type analysis, which automatically *infers* a safely approximated (in case of doubt, use "far") *location type* [47]. The inferred types are displayed in the Eclipse editor as superscripts ("N" for near, "F" for far) above the declared types. All annotations in the example can be inferred automatically: simply delete the annotations and save the file to see them. It is also possible to annotate a declaration with

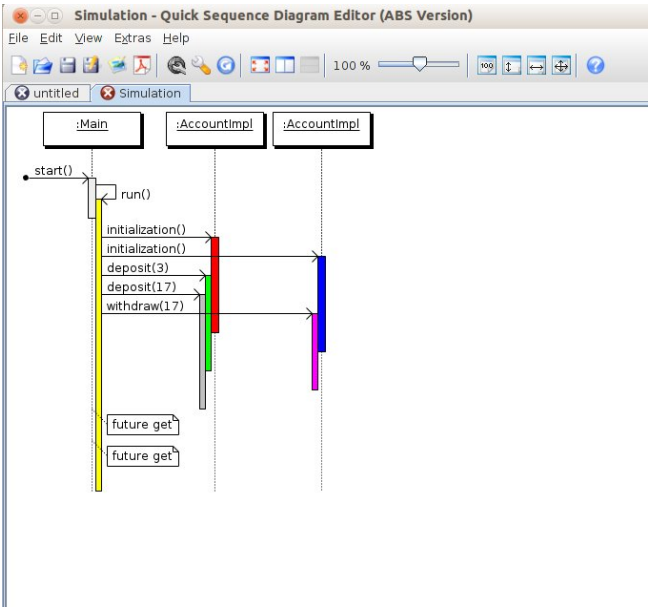


Fig. 9. Sequence diagram generated from an ABS simulation

`Somewhere`, which overrides the type inference mechanism and tells the compiler not to make any assumptions. Default is the annotation `Infer`, but this can be changed in the ABS project properties.

Deadlocks. It is entirely possible that execution of an ABS model results in a *deadlock* during runtime. A non-trivial example is provided by the model in Fig. 10, contained in file `Deadlock.abs`. Objects `c`, `e` and `d` reside in two different COGs, say the first two are in cog_c and the latter in cog_d. The task that executes `m1` is in cog_c while the task executing the call to `m2` inside `m1` is put into the second COG cog_d. During this execution `m3` is called on `e`, which is located in the first COG cog_c again. For `m3` to proceed the task needs to obtain the lock of cog_c, but this is not possible, because `m1` still waits for the result of `m2`. Hence, neither COG can progress.

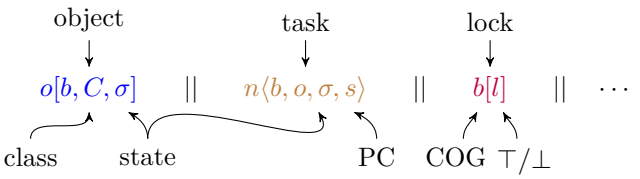
Deadlocks are very difficult to exclude in general. Deadlock-free concurrent languages tend to be too restrictive to be usable and, unlike data race-freeness, are not a practical option. In ABS many deadlocks can be avoided by supplying enough release points. In the example above it is sufficient to insert an `await` in front of one of the `get` expressions. In addition, there is an automatic deadlock analysis for ABS [23] that is currently being implemented.

```
class C {
  C m1(C b, C c) { Fut<C> r = b!m2(c); return r.get; }
  C m2(C c) { Fut<C> r = c!m3(); return r.get; }
  C m3() { return new C(); }
}
{
  C c = new C(); C d = new cog C(); C e = new C();
  c!m1(d,e);
}
```

Fig. 10. Example for deadlock in ABS

6.5 Formal Semantics of Concurrent ABS

The ABS language has a mathematically rigorous, SOS-style semantics [17,31]. This tutorial introduction is not the place go into the details, but we sketch the main ideas. The central issue is to give an appropriate structure to the terms that represent ABS *runtime configurations*. These are collections over the following items:



COGs are identified simply by a name b for their *lock* whose value can be either $\top$ or $\bot$.

Objects have a name o, need to have a reference to their COG b, to their class C, and they also have a local state σ that holds the current field values.

Tasks have a name n, a reference to their COG b and to the object o whose code they are executing. They also have a state σ with values of local variables and a *program counter* s that gives the next executable instruction. Task names n also double as futures, because they contain exactly the required information.

A runtime configuration may consist of any number of the above items. The operational semantics of ABS is given by rewrite rules that match the next executable statement of a task (and thereby also the current COG and object). A typical example is the rewrite rule that realizes *creation of a new COG*:

$$n\langle b, o, \sigma, T\ z = \textbf{new cog}\ C(v); s\rangle \rightarrow$$
$$b'(\top) \mid\mid n'\langle b', o', \sigma'_{init}, s_{task}\rangle \mid\mid o'[b', C, \sigma_{init}] \mid\mid n\langle b, o, \sigma, s\{z/o'\}\rangle$$

where:

- b', o', n' new;
- $\overline{Tf}; s'$ init block of class C and σ'_{init} binds constructor parameters v;
- $\sigma_{init} = \overline{Tf}$;
- $s_{task} = s'\{\mathbf{this}/o'; \texttt{suspend}\}$

The rule matches a **new** `cog` statement in task n, COG b, current object o, class C, and subsequent statements s. First we need to create a new COG with a fresh name b' and a new object o' in class C. The new COG starts immediately to execute the initialization code of its class C in a new task n', therefore, b''s lock is set to $\top$. Note that the current object **this** must be instantiated now to the actual object o'. After initialization, execution is suspended. The original task n immediately continues to execute the remaining code as there is no release point here. The value of the object reference z is replaced with the new object o'.

7 Extensions

7.1 Pluggable Type System

All declarations (fields, methods, classes, interfaces) in ABS can carry *annotations*. These are simply expressions that are enclosed in square brackets. The location type system in Sect. 6.4 provided examples. Other annotations can be logical expressions that are used as assertions, contracts, or invariants during verification or runtime assertion checking [14]. That goes beyond this tutorial.

The location types are a so-called *pluggable* type system. Such type systems can be realized easily in ABS via *meta annotations*. The special annotation `[TypeAnnotation]` declares the data type definition immediately following it to be a definition for type annotations and makes the parser aware of it. For example, the location type system is declared as follows:

```
[TypeAnnotation]
data LocationType = Far | Near | Somewhere | Infer;
// usage
[LocationType: Near] T n;
```

7.2 Foreign Language Interface

As a modeling language ABS does not contain mechanisms for I/O, because these are typically implementation-dependent. Of course, one often would like to have some output from the execution of an ABS model or connect an ABS model to existing legacy code. This is possible with a *foreign language interface* (FLI) mechanism that not only can be used to implement I/O for ABS, but to connect ABS models with legacy code in implementation languages in general.

At the moment, the ABS FLI is realized for the Java language. An ABS class that is to be implemented in Java needs three ingredients:

1. import of helper functions and classes from the module `ABS.FLI`;
2. a declaration as being foreign by the annotation `[Foreign]`;
3. default ABS implementations of all interface methods.

A simple example can look as follows:

```
import * from ABS.FLI;

interface Hello { String hello(String msg); }

[Foreign]
class HelloImpl implements Hello {
   String hello(String msg) { return "default implementation"; }
}

{
   Hello h = new HelloImpl();
   h.hello("Hi there");
}
```

The default implementation is used for simulation of ABS code without Java. It is now possible to implement a Java version of the `HelloImpl` class in a Java project and to connect that project with ABS. The details of how this is done are explained at the HATS tools site, see `http://tools.hats-project.eu/eclipseplugin/fli.html`. Basically, one extends the Java class `HelloImpl_c` that was generated by the ABS Java backend with a new implementation of the Java method `hello(String)`. By convention, the Java methods carry the prefix `fli`.

```
import abs.backend.java.lib.types.ABSString;
import abs.backend.java.lib.runtime.FLIHelper;
import Test.HelloImpl_c;

public class HelloImpl_fli extends HelloImpl_c {
   @Override
   public ABSString fli_hello(ABSString msg) {
      FLIHelper.println("I got "+msg.getString()+" from ABS");
      return ABSString.fromString("Hello ABS, this is Java");
   }
}
```

On the Java side any Java construct can be used. ABS provides a Java package `abs.backend.java.lib.types` containing declarations of the built-in ABS types usable in Java such as `ABSString`.

Execution of the ABS main block above will now cause the Java output “I got Hi there from ABS” to be printed on the system console.

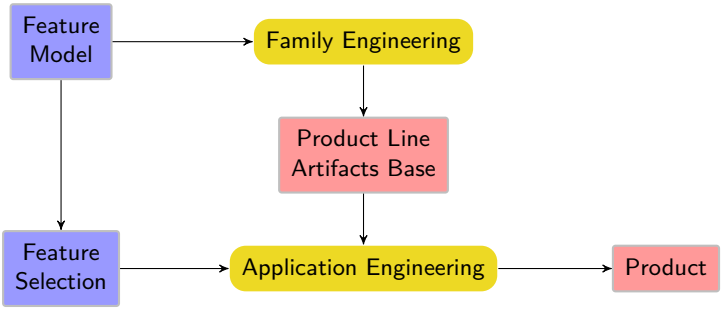


Fig. 11. Product line engineering

8 Product Line Modeling with ABS

8.1 Product Line Engineering

One of the aims of ABS is to provide a uniform and formal framework for *product line engineering* (PLE) [41], a practically highly successful software reuse methodology. In PLE one distinguishes two separate development phases (see Fig. 11).

During *family engineering* one attempts to distill the commonality among different products into a set of reusable artifacts. At the same time, the *variability* of the product line is carefully planned. This is typically done in a feature-driven manner, and the relation of features, as well as constraints in their combination is documented in a *feature model* with the help of a *feature description language* [45].

In the *application engineering* phase, individual products are being built by selecting features and by combining the artifacts that implement them in a suitable way.

One drawback of current practice in PLE is that feature description languages make no formal connection between features and their implementation. This renders products assembly ad hoc and error-prone. That issue is addressed in ABS with language extensions for modeling of features, for connecting features to their realization, as well as for feature selection and product specification [9]. In this section we introduce the PLE extensions of ABS. A fuller discussion of various approaches to achieve greater flexibility in object-oriented programming is contained in the chapter by van Dooren et al. in this volume [46] and in [16].

If one wants to maintain a connection between features and code, then the central issue are the mechanisms being used to compose the code corresponding to new features with the existing code. In current practice, this is often done by "glue code" written in scripting languages. ABS has the ambition that models can be statically analyzed. This means that the feature composition mechanism must be well-structured and represent a suitable match for the analysis methods used in ABS [18]. Such a mechanism is *delta-oriented programming* (DOP) [42,

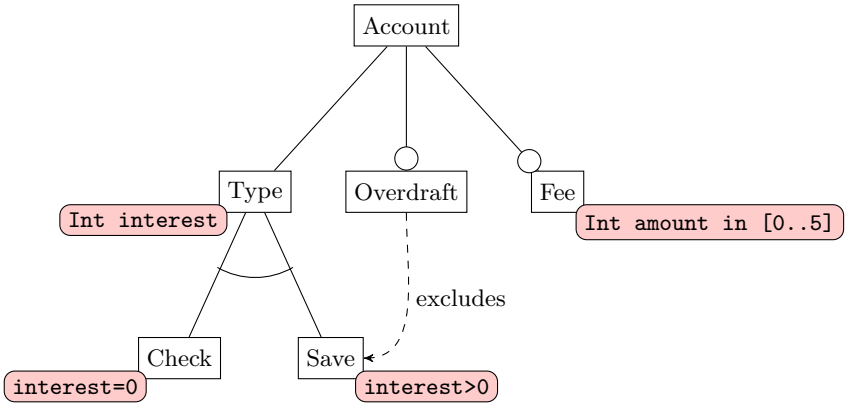


Fig. 12. Graphical representation of the `Account` feature model

43], because it allows to modify object-oriented code in a structured manner at the granularity of fields and methods, which is adequate for the contract-based specification and verification approach in ABS [26,27].

In brief, the ABS-extensions used to model product lines consist of four elements [8,9], which we describe now in turn:

1. A *feature description language*
2. A language for deltas that modify ABS models
3. A configuration language connecting features with the deltas that realize them
4. A language for *product configuration*

8.2 Feature Description

Modern software development processes, notably agile processes and PLE, tend to be feature-driven. A number of mature and useful formalisms for feature description have been developed in the last decade. For ABS we use a slight modification of the *textual variability language* (TVL) [10], which has the advantage of having a formal semantics and a textual representation. The flavour of this language used in ABS is called μTVL and differs from TVL in that (i) attribute types that are not needed are omitted and (ii) the possibility to have multiple root features. The latter is useful to model orthogonal variability in product lines.

Let us build a product line based on the `Account` interface from Sect. 6.4. Assume we want to distinguish between checking and saving accounts. The latter may pay interest, whereas the former usually don't. Optionally, a checking account (but not a saving account) may permit an overdraft or incur fees for transactions. A graphical representation of the `Account` feature model is in Fig. 12. The textual rendering in μTVL in file `FeatureModel.abs` looks as follows:

```
root Account {
  group allof {
    Type {
      group oneof {
        Check {ifin: Type.i == 0;},
        Save {ifin: Type.i > 0;
              exclude: Overdraft;}
      }
      Int i; // interest rate
    },
    opt Fee {Int amount in [0..5];},
    opt Overdraft
  }
}
```

In μTVL one represents each subhierarchy in the feature tree by a `group` of features, which can be further qualified as inclusive (`allof`) or alternative (`oneof`). Within a group there is a comma-separated list of feature declarations. Each feature declaration may be optional (`opt`) and have restrictions (`ifin:`), exclusions (`exclude:`), or requirements (`include:`). Feature parameters are declared *after* the declaration of a subhierarchy. A feature model appears in a separate file with extension `.abs`. The ECLIPSE editor supports syntax and parse error highlighting. There can be several feature files with feature declarations. These are interpreted as orthogonal feature hierarchies that are all part of the same feature model.

The semantics of feature models is straightforward by translation into a boolean/integer constraint formula, see [9, 10]. For example, the feature model above is characterized by the following formula:

$$
\begin{aligned}
&0 \leq \texttt{Account} \leq 1 \ \wedge \ \texttt{Type} \rightarrow \texttt{Account} \ \wedge \\
&\texttt{Overdraft}^{\dagger} \rightarrow \texttt{Account} \ \wedge \ \texttt{Fee}^{\dagger} \rightarrow \texttt{Account} \ \wedge \\
&\texttt{Type} + \texttt{Fee}^{\dagger} + \texttt{Overdraft}^{\dagger} = 3 \ \wedge \ 0 \leq \texttt{Type} \leq 1 \ \wedge \\
&\texttt{Check} \rightarrow \texttt{Type} \ \wedge \ \texttt{Save} \rightarrow \texttt{Type} \ \wedge \ \texttt{Save} \rightarrow \neg\texttt{Overdraft} \wedge \\
&\texttt{Check} + \texttt{Save} = 1 \ \wedge \\
&0 \leq \texttt{Check} \leq 1 \ \wedge \ 0 \leq \texttt{Save} \leq 1 \ \wedge \ 0 \leq \texttt{Fee}^{\dagger} \leq 1 \ \wedge \ 0 \leq \texttt{Overdraft}^{\dagger} \leq 1 \wedge \\
&\texttt{Fee} \rightarrow \texttt{Fee}^{\dagger} \ \wedge \ \texttt{Overdraft} \rightarrow \texttt{Overdraft}^{\dagger} \wedge \\
&0 \leq \texttt{Save} \leq 1 \ \wedge \ 0 \leq \texttt{Check} \leq 1 \ \wedge \\
&\texttt{Fee} \rightarrow (\texttt{Fee.amount} >= 0 \ \wedge \ \texttt{Fee.amount} <= 5) \ \wedge \\
&\texttt{Check} \rightarrow (\texttt{Type.i} = 0) \ \wedge \ \texttt{Save} \rightarrow (\texttt{Type.i} > 0).
\end{aligned}
$$

Features are represented by integer variables of the same name. We use the convention that these can also be used as boolean variables where a non-0 value denotes *true*. Optional features are represented by two variables, one of which is labeled with †. The latter denotes the case when the feature is actually selected.

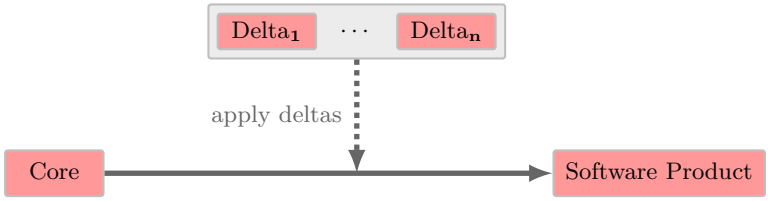


Fig. 13. Application of delta modules to a core product

This is formalized in expressions such as "$\texttt{Type} + \texttt{Fee}^{\dagger} + \texttt{Overdraft}^{\dagger} = 3$". Some constraints are redundant, for example, it would be sufficient to write "$\texttt{Account} \leq 1$", but checking for such situations complicates the translation function unnecessarily.

It is easy to check validity of a given feature selection for a feature model F: for any selected feature f and parameter value $p := v$ one simply conjoins the formula $f = 1 \wedge p = v$ to the semantics of F and checks for satisfiability with a constraint solver. The constraint solver of ABS can:

- find all possible solutions for a given feature model and
- check whether a feature selection is a solution of a feature model.

The latter check is performed implicitly in the ABS Eclipse plugin, whenever the user requests to build a product based on a specific feature selection (see Sect. 8.5).

8.3 Delta Modeling

As mentioned above, the realization of features in ABS is done with *delta modules* (or deltas, for short), a variant of delta-oriented programming (DOP). This constitutes the main *reuse principle* of ABS and replaces other mechanisms such as code inheritance, traits, or mixins. Specifically, in standard OO languages the dynamic selection of software components is often realized by *late binding* mechanisms such as dynamic dispatch. But preserving type-safety in such a context means to encode both, variability and data modeling into the subtyping relation. This can lead to convoluted and brittle type hierarchies. In ABS we decided to achieve a strict separation of concerns between data modeling and variability modeling by employing the dedicated syntactic mechanism of delta modules for the latter. For a detailed discussion of the advantages of DOP for variability modeling see [16]. Recently it has been shown [26,27] that delta-oriented modeling is also a viable basis for modular specification and verification of feature-rich software.

In delta modeling we assume that one outcome of the family engineering phase (see Fig. 11) is a *core* or *base product* with minimal functionality. Product variants with additional features are obtained from it by applying one or more

deltas that realize the desired features, as illustrated in Fig. 13. In ABS, deltas have the following capabilities:

- Delta modules may *add*, *remove* or *modify* classes and interfaces
- Permitted class modifications are:
  - adding and removal of *fields*
  - adding, removal and modification of *methods*
  - extending the list of implemented interfaces

The actual reuse mechanism is located in the modification of methods: the description of a method modification in a delta can access the most recent incarnation of that method in a previous delta by the statement `original(...);`. This will cause the compiler to insert the body of the referred method at the time when the deltas are applied. The signature of `original()` must be identical to the one of the modified method. The compiler checks the applicability of deltas and ensures well-typedness of the resulting code. It is because of this reuse mechanism that once can say that the *granularity* of delta application is at the level of methods.

There is a certain analogy between `original()` in DOP and **super**()-calls in OO languages with code inheritance. The crucial difference is that `original()` references are resolved at compile time (product build time), while **super**()-calls occur at runtime. As a consequence, there is a runtime penalty for the latter. In addition, the semantics of **super**() is more complex than that of `original()`, because it interacts with dynamic method dispatch.

Assume we have the following implementation of the `withdraw(Int)` method of the `Account` interface, which ensures that we cannot withdraw more than the current balance:

```
class AccountImpl(Int aid, Int balance, Customer owner)
      implements Account {
  Int withdraw(Int x) {
    if (balance - x >= 0) { balance = balance - x; }
    return balance;
  }
}
```

Now we would like to create a delta module that realizes the feature `Fee`. We need to modify `withdraw(Int)`, which can be achieved by the following delta, contained in file `Fee.abs`:

```
delta DFee(Int fee); // Implements feature Fee
uses Account;
modifies class AccountImpl {
  modifies Int withdraw(Int x) {
    Int result = x;
    if (x>=fee) result = original(x+fee);
    return result;
  }
}
```

The modified `withdraw(Int)` method is implemented by a suitable call to the original version after a check that the withdrawn amount is not trivially small. The new implementation is a "wrapper" around the original one, which is a typical idiom in DOP. Of course, it is also possible to replace a method completely. In ABS method deltas one must usually declare a `result` variable to ensure that the **return** statement is last, as can be seen here.

One or more deltas can be placed into a file with extension `.abs`. The connection between different deltas and a base implementation is given via the `uses` clause that refers to the module where the base is found. Like classes, deltas can have parameters, however, these are not fields, but are instantiated at product build time. Normally, there is a correspondence between the parameters of deltas and those of the features they are supposed to implement.

Assume further we want to realize the `Save` feature. One must ensure that the interest rate is set to 0. ABS deltas at this time do not support to add or modify class initialization blocks. To change the initial value of a field, we simply remove the field declaration and add it again with a suitable initial value:

```
delta DSave(Int i); // Implements feature Save
uses Account;
modifies class AccountImpl {
  removes Int interest;
  adds Int interest = i;
}
```

Of course, we assume here that the `interest` field has been added in the first place in the earlier delta `DType`. This requires to specify and check temporal constraints on the application of deltas as shall be seen in the following section. Application of a concrete delta is illustrated with `DSave` in Fig. 14.

Syntax and parse error highlighting for delta modules works as usual. Automatic completion works as well, but it is only done relative to the base product. The reason is that before product build time, the compiler cannot know which deltas have been applied before. For the same reason, only a limited amount of type checking is done. Research to lift type checking to the family level is under way [12, 36].

8.4 Product Line Configuration

So far, we have two models relating to product lines: the *feature model* and the *delta model*, that is, the feature implementation. Unlike any other formalism we are aware of, in ABS we can make a formal connection between these.[5] This is the key to being able to analyze implementations of whole *product lines* and not merely individual products.

In ABS, the connection between features and their realization (illustrated in Fig. 15) is done in a dedicated *product line configuration* file. This makes

[5] Model checking approaches to PLE such as [11] attach behavior to features, but make a finite state abstraction in the form of labeled transition systems.

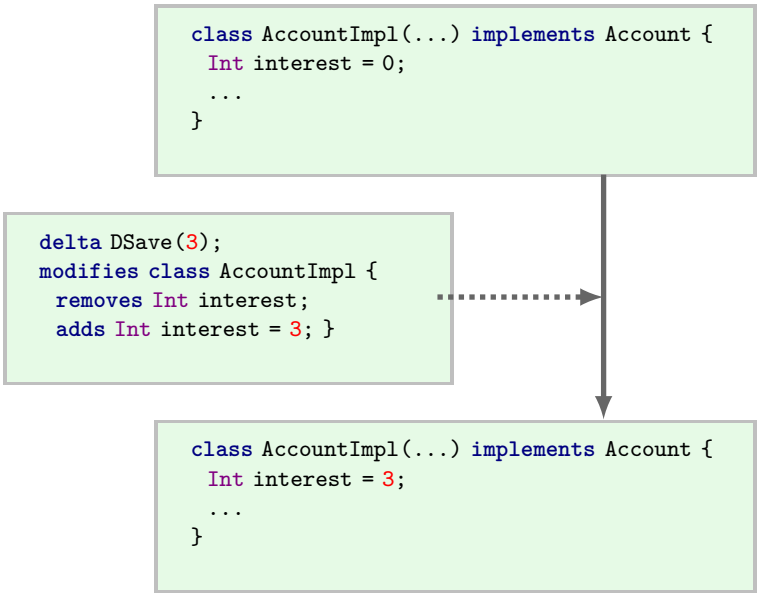


Fig. 14. Application of delta `DSave`

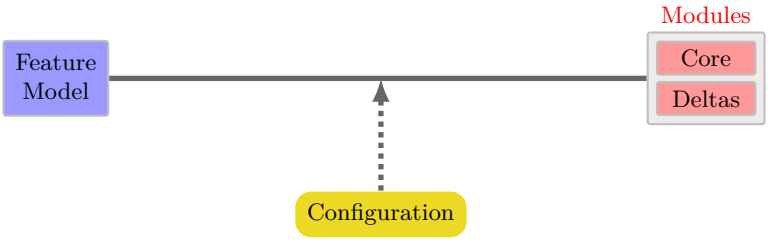


Fig. 15. Schema of product line configuration in ABS

debugging easy, because all information about the realization of a given feature model is collected in a single location. To establish a connection between features and deltas, the configuration files need to specify three aspects:

1. they must associate features with their implementing delta modules by *application conditions*;
2. they need to resolve conflicts in the application order by giving partial *temporal constraints* on delta application;
3. they need to pass the *attribute values* of features to the parameters of the delta modules.

We can illustrate all three aspects with our running example. The following file `AccountPL.abs` defines a product line named `Accounts` based on the five features of the feature model in Fig. 12.

```
productline Accounts;
features Type, Fee, Overdraft, Check, Save;

delta DType (Type.i) when Type;
delta DFee (Fee.amount) when Fee;
delta DOverdraft after DCheck when Overdraft;
delta DSave (Type.i) after DType when Save;
delta DCheck after DType when Check;
```

For each delta that is to be used for implementing any of the features one specifies:

- the application conditions (`when` clauses), that is, the feature(s) that are being realized by each delta and whose presence triggers delta application;
- the delta parameters which are derived from feature attribute values;
- a strict partial order of delta application (`after` clauses) to ensure well-definedness of delta applications and resolve conflicts.

In the example, there is a one-to-one correspondence between deltas and features, which is reflected in the application conditions. Likewise, the feature attributes `Type.i` and `Fee.amount` directly can be used as parameters of the corresponding deltas. The temporal constraints of `DSave` and `DCheck` ensure that the field `interest` is present. The constraint of `DOverdraft` makes sure that this delta is only applied to checking accounts. It would also have been possible to express this constraint at the level of the feature model with an `includes:` clause. It is up to the modeler to decide whether a given constraint is a property of the feature model or of the product line.

8.5 Product Selection

The final step in PLE with ABS is *product selection*. Whereas the activities that took place until now can be viewed as mostly being part of family engineering, the selection process is always part of application engineering.

To create a product it is sufficient to list the features that should be realized in it and to instantiate the feature attributes with concrete values. The syntax is very simple and self-explaining. As any other ABS file, product selection files have the `.abs` extension and there is Eclipse support for syntax and parse error highlighting. Some examples for the `Accounts` product line are given in the file `Products.abs` as follows:

```
product CheckingAccount (Type{i=0},Check);
product AccountWithFee (Type{i=0},Check,Fee{amount=1});
product AccountWithOverdraft (Type{i=0},Check,Overdraft);
product SavingWithOverdraft (Type{i=1},Save,Overdraft);
```

The simplest product that can be built is `CheckingAccount`. The second product above extends it by charging a fee of one unit per transaction. The ABS compiler

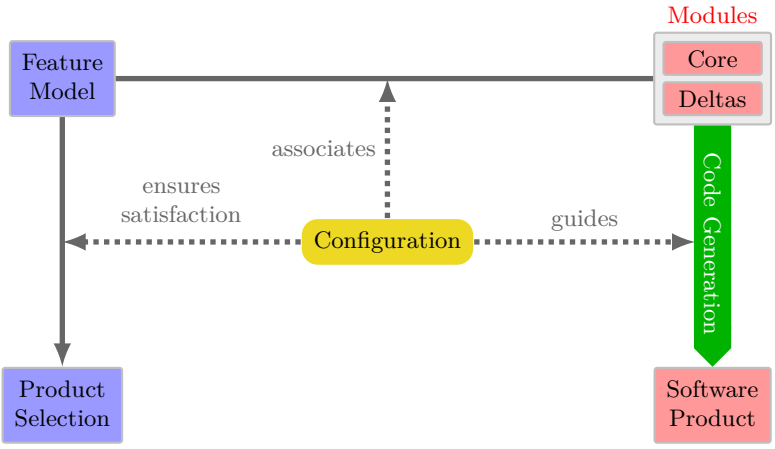


Fig. 16. The role of product line configuration in product selection and compilation

uses the product selection file and the other related files to create a "flattened" ABS model where all deltas have been applied such that it contains only core ABS code. In a first step, the compiler checks that the product selection is valid for the given feature model as described in Sect. 8.2. It then uses the product line configuration file to decide which deltas need to be applied and how they are instantiated. The partial order on the deltas is linearized. It is up to the modeler to ensure (possibly, by adding temporal constraints) that different linearizations do not lead to conflicting results. Finally, the resulting core ABS model is type-checked and compiled to one of the ABS backends in the standard way. As all parts of the ABS language the product line modeling languages have a formal semantics—the details are found in [9].

The generation of a declared product can be triggered in Eclipse from the Run|Run Configurations ... dialog by creating a new ABS Java Backend configuration (or using an existing one). In the ABS Java Backend tab the available products appear under the ABS Product menu. Invalid product selections or type errors in the configuration files will be displayed at this stage. For example, selection of the `SavingWithOverdraft` product above results in an error, because the constraints in the feature model are not satisfied. After selection of a valid product one can run and debug the resulting core ABS model as described earlier. The ABS compiler additionally creates always a base product that corresponds to the given ABS model without any features or deltas. This product appears under the name `<base>` in the product selection menu.

If we execute the main class of the `Account` module in Sect. 6.4 in the base product, we obtain the result 37 in the variable `net`, whereas if we run the product `AccountWithFee`, we obtain 34.

A current limitation of the Eclipse ABS plugin is that the debugger correctly displays the runtime configuration and the values of variables of products, but in the editor window only the core product is displayed, not the actual product with applied deltas.

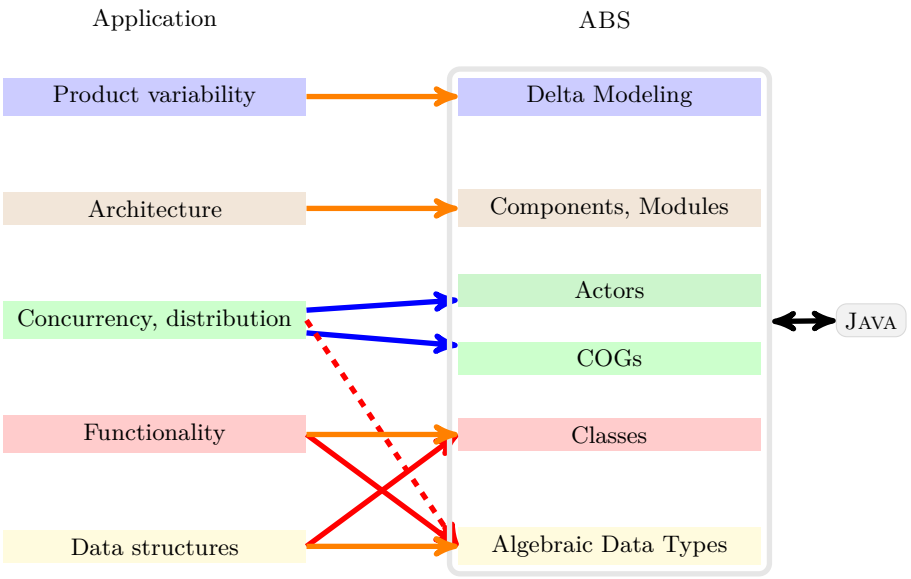


Fig. 17. Choice of different layers when modeling with ABS

9 Concluding Remarks

In this tutorial we gave an introduction to the abstract modeling language ABS. Uniquely among current modeling languages, ABS has a formal semantics *and* covers the whole spectrum from feature modeling to the generation of executable code in Java. Development and analysis of ABS models are supported by an Eclipse plugin.

A very important point is that ABS offers a wide variety of modeling options in a uniform, homogeneous framework, see Fig. 17. This allows the modeler to select an appropriate style and level of abstraction for each modeled artifact. For example, data structures and functionality of sequential methods can be modeled either in functional or in OO style (red arrows). The possibility to use ADTs enables rapid prototyping and design-time analysis. For example, concurrent behavior may first be abstracted into an ADT and only refined later (dashed arrow). All language layers of ABS are integrated in a homogeneous framework and have a uniform formal semantics. Mixing ABS with legacy code or deployment-specific functionality such as I/O is possible via the Java FLI.

Of course, as any other formalism, ABS has also limitations: it is not designed to model low-level, thread-based concurrency with shared data. Hence, ABS in its current state is not suitable to model multi-core applications or system libraries. In this sense, approaches to formalization of low-level concurrency [5] are complementary to ABS.

As mentioned earlier, the analysis capabilities and the ABS runtime component layer are beyond this tutorial, but some chapters in this volume cover part of that material.

Acknowledgments. The development and implementation of ABS was a collaborative effort of the many researchers involved in the HATS project. While the text of this tutorial has been written from scratch by the author, it could never have been done without the background of all the papers, presentations, and discussions provided by many colleagues. Special thanks go to the main contributors to Work Package 8: Frank de Boer, Einar Broch Johnsen, and Ina Schaefer.

The hints from two anonymous reviewers led to a number of improvements and clarifications.

References

[1] Abrial, J.-R.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010)
[2] The ABS Language Specification, ABS version 1.2.0 edn. (April 2013), `http://tools.hats-project.eu/download/absrefmanual.pdf`.
[3] Ahrendt, W., Dylla, M.: A system for compositional verification of asynchronous objects. Science of Computer Programming 77(12), 1289–1309 (2012)
[4] Albert, E., et al.: Automatic inference of bounds on resource consumption. In: de Boer, F., Bonsangue, M., Giachino, E., Hähnle, R. (eds.) FMCO 2012. LNCS, vol. 7866, pp. 119–144. Springer, Heidelberg (2013)
[5] Amighi, A., Blom, S., Huisman, M., Zaharieva-Stojanovski, M.: The VerCors project: setting up basecamp. In: Claessen, K., Swamy, N. (eds.) Proc. Sixth Workshop on Programming Languages Meets Program Verification, PLPV, Philadelphia, PA, USA, pp. 71–82. ACM (2012)
[6] Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
[7] Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer (2003)
[8] Clarke, D., Diakov, N., Hähnle, R., Johnsen, E.B., Schaefer, I., Schäfer, J., Schlatte, R., Wong, P.Y.H.: Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 417–457. Springer, Heidelberg (2011)
[9] Clarke, D., Muschevici, R., Proença, J., Schaefer, I., Schlatte, R.: Variability modelling in the ABS language. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 204–224. Springer, Heidelberg (2011)
[10] Classen, A., Boucher, Q., Heymans, P.: A text-based approach to feature modelling: Syntax and semantics of TVL. Science of Computer Programming 76(12), 1130–1143 (2011)
[11] Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.-Y.: Model checking software product lines with SNIP. International Journal on Software Tools for Technology Transfer (STTT) 14, 589–612 (2012)
[12] Damiani, F., Schaefer, I.: Family-based analysis of type safety for delta-oriented software product lines. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012, Part I. LNCS, vol. 7609, pp. 193–207. Springer, Heidelberg (2012)

[13] de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
[14] de Boer, F.S., de Gouw, S.: Run-time verification of black-box components using behavioral specifications: An experience report on tool development. In: Păsăreanu, C.S., Salaün, G. (eds.) FACS 2012. LNCS, vol. 7684, pp. 128–133. Springer, Heidelberg (2013)
[15] Report on the Core ABS Language and Methodology: Part A, Part of Deliverable 1.1 of project FP7-231620 (HATS) (March 2010), http://www.hats-project.eu
[16] Final Report on Feature Selection and Integration, Deliverable 2.2b of project FP7-231620 (HATS) (March 2011), http://www.hats-project.eu
[17] Full ABS Modeling Framework, Deliverable 1.2 of project FP7-231620 (HATS) (March 2011), http://www.hats-project.eu
[18] Analysis Final Report, Deliverable 2.7 of project FP7-231620 (HATS) (December 2012), http://www.hats-project.eu
[19] Evaluation of Modeling, Deliverable 5.3 of project FP7-231620 (HATS) (March 2012), http://www.hats-project.eu
[20] Model Mining, Deliverable 3.2 of project FP7-231620 (HATS) (March 2012), http://www.hats-project.eu
[21] Correctness, Deliverable 4.3 of project FP7-231620 (HATS) (March 2013), http://www.hats-project.eu
[22] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
[23] Giachino, E., Laneve, C.: Analysis of deadlocks in object groups. In: Bruni, R., Dingel, J. (eds.) FORTE 2011 and FMOODS 2011. LNCS, vol. 6722, pp. 168–182. Springer, Heidelberg (2011)
[24] Girault, C., Valk, R.: Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications. Springer, Secaucus (2001)
[25] Hähnle, R., Helvensteijn, M., Johnsen, E.B., Lienhardt, M., Sangiorgi, D., Schaefer, I., Wong, P.Y.H.: HATS abstract behavioral specification: the architectural view. In: Beckert, B., Damiani, F., de Boer, F., Bonsangue, M.M. (eds.) FMCO 2011. LNCS, vol. 7542, pp. 109–132. Springer, Heidelberg (2013)
[26] Hähnle, R., Schaefer, I.: A Liskov principle for delta-oriented programming. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012, Part I. LNCS, vol. 7609, pp. 32–46. Springer, Heidelberg (2012)
[27] Hähnle, R., Schaefer, I., Bubel, R.: Reuse in software verification by abstract method calls. In: Bonacina, M.P. (ed.) CADE 2013. LNCS, vol. 7898, pp. 300–314. Springer, Heidelberg (2013)
[28] Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8(3), 231–274 (1987)
[29] Hewitt, C., Bishop, P., Steiger, R.: A universal modular ACTOR formalism for Artificial Intelligence. In: Nilsson, N.J. (ed.) Proc. 3rd International Joint Conference on Artificial Intelligence, pp. 235–245. William Kaufmann, Stanford (1973)
[30] Johnsen, E.B.: Separating cost and capacity for load balancing in ABS deployment models. In: de Boer, F., Bonsangue, M., Giachino, E., Hähnle, R. (eds.) FMCO 2012. LNCS, vol. 7866, pp. 145–167. Springer, Heidelberg (2013)
[31] Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)

[32] Johnsen, E.B., Owe, O., Yu, I.C.: Creol: A type-safe object-oriented model for distributed concurrent systems. Theoretical Computer Science 365(1-2), 23–66 (2006)
[33] Kurnia, I.W., Poetzsch-Heffter, A.: Verification of open concurrent object systems. In: de Boer, F., Bonsangue, M., Giachino, E., Hähnle, R. (eds.) FMCO 2012. LNCS, vol. 7866, pp. 83–118. Springer, Heidelberg (2013)
[34] Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M.: JML Reference Manual. Draft revision 1.235 (September 2009)
[35] Lienhardt, M., Bravetti, M., Sangiorgi, D.: An object group-based component model. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012, Part I. LNCS, vol. 7609, pp. 64–78. Springer, Heidelberg (2012)
[36] Lienhardt, M., Clarke, D.: Conflict detection in delta-oriented programming. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012, Part I. LNCS, vol. 7609, pp. 178–192. Springer, Heidelberg (2012)
[37] Milne, G.: Design for verifiability. In: Leeser, M., Brown, G. (eds.) Hardware Specification, Verification and Synthesis: Mathematical Aspects. LNCS, vol. 408, pp. 1–13. Springer, Heidelberg (1990)
[38] Milner, R., Parrow, J., Walker, J.: A calculus of mobile processes, I and II. Inform. and Comput. 100(1), 1–40, 41–77 (1992)
[39] Nobakht, B., de Boer, F.S., Jaghoori, M.M., Schlatte, R.: Programming and deployment of active objects with application-level scheduling. In: Proceedings of the 2012 ACM Symposium on Applied Computing (SAC). ACM (2012)
[40] Peyton Jones, S. (ed.): Haskell 98 Language and Libraries: The Revised Report (September 2002), `http://haskell.org/`
[41] Pohl, K., Böckle, G., Van Der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer (2005)
[42] Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 77–91. Springer, Heidelberg (2010)
[43] Schaefer, I., Worret, A., Poetzsch-Heffter, A.: A Model-Based Framework for Automated Product Derivation. In: Proc. of Workshop in Model-based Approaches for Product Line Engineering, MAPLE 2009 (2009)
[44] Schäfer, J., Poetzsch-Heffter, A.: Jcobox: Generalizing active objects to concurrent components. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 275–299. Springer, Heidelberg (2010)
[45] Schobbens, P., Heymans, P., Trigaux, J.: Feature diagrams: A survey and a formal semantics. In: 14th IEEE International Conference on Requirements Engineering, pp. 139–148 (2006)
[46] van Dooren, M., Clarke, D., Jacobs, B.: Subobject-Oriented programming. In: de Boer, F., Bonsangue, M., Giachino, E., Hähnle, R. (eds.) FMCO 2012. LNCS, vol. 7866, pp. 38–82. Springer, Heidelberg (2013)
[47] Welsch, Y., Schäfer, J.: Location types for safe distributed object-oriented programming. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 194–210. Springer, Heidelberg (2011)
[48] Wong, P.Y.H., Albert, E., Muschevici, R., Proença, J., Schäfer, J., Schlatte, R.: The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. Journal on Software Tools for Technology Transfer 14(5), 567–588 (2012)

Subobject-Oriented Programming

Marko van Dooren, Dave Clarke, and Bart Jacobs

iMinds-DistriNet, Department of Computer Science,
KU Leuven, Belgium
firstname.lastname@cs.kuleuven.be

Abstract. Classes are fundamental elements in object-oriented programming, but they cannot be assembled in a truly flexible manner from other classes. As a result, cross-cutting structural code for implementing associations, graph structures, and so forth must be implemented over and over again. Subobject-oriented programming overcomes this problem by augmenting object-oriented programming with subobjects. Subobjects can be used as buildings blocks to configure and compose classes without suffering from name conflicts. This paper gives an overview of subobject-oriented programming and introduces mechanisms for subobject initialization, navigation of the subobject structure in super calls, and subobject refinement. Subobject-oriented programming has been implemented as a language extension to Java with Eclipse support and as a library in Python.

1 Introduction

Class-based object-oriented programming enables programmers to write code in terms of abstractions defined using classes and objects. Classes are constructed from *building blocks*, which are typically other classes, using inheritance. This allows a new class to extend and override the functionality of an existing class. More advanced mechanisms such as multiple inheritance, mixins, and traits extend the basic inheritance mechanism to improve code reuse. Although becoming increasingly popular in mainstream languages, these mechanisms suffer from a number of problems: code repetition, conceptually inconsistent hierarchies, the need for glue code, ambiguity, and name conflicts.

Code repetition occurs because code cannot be sufficiently modularized in single and multiple inheritance hierarchies. For instance, many classes are based on high-level concepts such as associations (uni- or bi-directional relationships between objects) or graph structures, for example, in classes representing road networks. When implementing such classes, current best practice is to implement these high-level concepts with lots of cross-cutting, low-level boilerplate code. As a result, what is essentially the same code, is written over and over again. Attempts to improve reuse often result in conceptually inconsistent hierarchies, in which semantically unrelated code is placed in classes just to enable reuse. Often reuse is achieved by abandoning inheritance and instead delegating to auxiliary objects (delegates) that implement the desired functionality. This requires additional glue code to put the delegates in place and to initialise them. Even more glue code is required to handle overriding of delegate functionality in subclasses. Inheritance mechanisms that enable inheritance from multiple sources, such as multiple inheritance, mixins, and traits, cannot properly deal with methods that have the same name but come from different super classes or traits.

E. Giachino et al. (Eds.): FMCO 2012, LNCS 7866, pp. 38–82, 2013.

Modern programming languages offer some solutions to these problems, but these are inadequate for various reasons (see Section 2 for more detail). Composition mechanisms such as mixins [4] and traits [21,19,8,7] cannot be used to turn the low-level code into reusable building blocks, because only one building block of a given kind can be used in a class. With non-virtual inheritance in C++ [22], the composer has no control over the inheritance hierarchy of the building block, and methods cannot always be overridden. With non-conformant inheritance in Eiffel [23], all methods of all building blocks must be separated individually. This requires a large amount of work and is error-prone. Aspect-oriented programming [15] does modularize cross-cutting code, but the cross-cutting code is of a different nature. In current aspect-oriented languages, the cross-cutting code of aspects augments the basic functionality of a system, whereas in the case of e.g. the graph functionality for road networks, the cross-cutting code defines the basic structure of the classes. While features such as intertype declarations in AspectJ [14] and wrappers in CaesarJ [1] can change the class structure, their purpose is to support aspects. As such, they do not provide the expressiveness to capture structural patterns that can be added multiple times to a single class, which is needed for a building block approach. And even if such expressiveness were supported, a class should not have to rely on aspects to define its basic structure. Together these problems indicate that the currently available building blocks for constructing classes are inadequate.

Subobject-oriented programming, the class composition mechanism described in this paper, addresses the limitations described above. Subobject-oriented programming augments object-oriented programming with subobjects that allow classes to be flexibly used as building blocks to create other classes. Subobjects do not introduce name conflicts, as different subobjects are isolated by default and can only interact after configuration in the composed class. Subobject members are truly integrated into the composed class and can be redefined in subclasses of the composed class. Subobject-oriented programming improves the *component relation* developed in previous work [26]; it has been used to make software transactional memory more transparent [24]. In this paper, we present an overview of subobject-oriented programming, along with the following new contributions:

- Language constructs for subobject initialization and navigation of the subobject structure in super calls.
- A simplified, object-oriented, approach to subobject configuration.
- A further binding mechanism to refine subobjects in a subclass.
- A formal models of the semantics of subobjects.
- Experimental validation: we have implemented subobject-oriented programming as a Java [12] extension called JLo [25], and as a Python 3 [20] library, and developed an extended example demonstrating how graph functionality can be implemented top of an existing association mechanism.

Organization: Section 2 presents an evaluation of existing reuse mechanisms. Section 3 defines subobject-oriented programming, including subobject customization and initialization. Section 4 shows additional examples of subobject-oriented programming. Section 5 discusses the implementation. Section 6 presents a semantic model of subobject-oriented programming, focusing on the semantics of method dispatch. Section 7 discusses related work, and Section 8 concludes.

2 Evaluation of Current Reuse Mechanisms

This section evaluates existing reuse techniques by trying to compose a simple class of radios from generic building blocks. The interface of class *Radio* is shown in Figure 1(a). The volume of a radio is an integer between 0 and 11, and the frequency is a float between 87.5 and 108.

```
class Radio {
        Radio(int, float)
  int   getVolume()
  void  setVolume(int)
  float getFrequency()
  void  setFrequency(float)
}
```

(a) The interface of *Radio*

```
class BoundedValue
  <T extends Comparable<T>> {
       BoundedValue(T, T, T)
  T    getMax()
  T    getValue()
  void setValue(T)
  T    getMin()
}
```

(b) The interface of *BoundedValue*

(c) A user interface for a radio

```
class Slider {
  connect(BoundedValue value)
  ...
}
```

(d) The interface of *Slider*

Fig. 1. The radio example

The behavior of both the volume and frequency is similar in that both are numeric values that must remain within certain bounds. This behavior is encapsulated in *BoundedValue*, whose interface is shown in Figure 1(b). We want to implement *Radio* using two building blocks of this class. Note that *BoundedValue* itself could be implemented with three building blocks for the value and the upper and lower bounds, but we do not consider nested composition for this example as it does not add to the discussion.

To make a user interface for the radio, we want to use two *Slider* objects and connect them to the frequency and the volume building blocks *as if they were* separate objects.

The resulting class *Radio* should offer at least the same functionality as if it were implemented from scratch. External clients of *Radio* should see it via an interface similar to the one in Figure 1(a), and subclasses of *Radio* should be able to redefine its methods.

2.1 Aspect-Oriented Programming

In the original paper on aspect-orientation [15], aspects captured any kind of cross-cutting code. Current aspect-oriented languages, however, focus on aspects that modify the behavior of existing code. Although aspect-oriented languages provide features to modify the structure of an existing class, such as intertype declarations in AspectJ [14]

and wrappers in CaesarJ [1], they do not provide the expressiveness that is needed to add them multiple times to a single class. As such, the bounded values of *Radio* cannot be added with aspects.

But even if the required expressiveness would be added, the nature of the cross-cutting code makes aspect-oriented programming inappropriate for the job. In such an aspect-oriented language, the volume and frequency of the radio would be added to *Radio* in a separate aspect. The bounded values for the volume and the frequency, however, define the basic behavior of a radio, and should therefore be declared in class *Radio*.

2.2 Mixins

Mixin inheritance [4] provides a limited form of multiple inheritance. A mixin is a class with an abstract superclass. This super class can be instantiated differently in different contexts. A concrete class specifies its inheritance hierarchy by declaring a list of mixins, which is called a mixin composition. The mixin composition linearizes the inheritance hierarchy of a class by using the successor of each mixin in the list as the super class for that mixin. By eliminating diamond structures, the linearization mechanism automatically solves name conflicts. This, however, can cause a method to override another method with the same signature by accident.

Fig. 2 shows a Scala [18][1] example where we attempt to reuse class *BoundedValue* twice in class *Radio*. Class *BoundedValue* itself is omitted. Because of the linearization, it is impossible to use mixins to implement the volume and frequency of a radio in this fashion. All methods inherited through the clause ***with*** *BoundedValue[Float]* would be overridden by or cause conflicts with the (same) methods inherited through the clause ***with*** *BoundedValue[Integer]*.

```
class Radio extends BoundedValue[Int] with BoundedValue[Float]
```

Fig. 2. An attempt to implement *Radio* with mixins

2.3 Traits

A trait [21] is a reusable unit of behavior that consist of a set of methods. A class uses a trait as a building block through *trait composition*, and resolves name conflicts explicitly using trait operators such as *alias* or *exclude*. The *flattening property* of traits states that the behavior of a trait composition is the same as if the trait code was written in the composing class. Lexically nested traits [7] improve upon the original traits by allowing a trait to have state and providing a lexical mechanism for controlling the visibility of the methods provided by a trait. Lexically nested traits are implemented in the AmbientTalk [7] programming language. For the sake of simplicity, the example code uses a hypothetical language TraitJava, which adds lexically nested traits to Java.

[1] Note that Scala does not support traits; it uses the keyword *trait* for mixins.

Fig. 3 shows an attempt to implement a radio using trait composition in TraitJava. Trait composition is done by *using* the trait. Class *Radio* uses two *BoundedValue* traits to model the volume and the frequency. Aliasing is used to resolve the name conflicts caused by using two traits with methods that have the same name.

```
class Radio {
  uses BoundedValue<Int> {
    alias value -> volume, max -> maxVolume,
          min -> minVolume, setValue -> setVolume
  }
  uses BoundedValue<Float> {
    alias value -> frequency, max -> maxFrequency,
          min -> minFrequency, setValue -> setFrequency
  }
}
trait BoundedValue<T> {
  T _value;
  T value() {return _value;}
  void setValue(T t) {
    if(t >= min() && t <= max()) {_value = t;}
  }
  T max() {...}
  T min() {...}
  ...
}
```

Fig. 3. Traits in the hypothetical language TraitJava

While at first sight this code seems to do what we need, it does not actually work. The fields of both bounded values are properly isolated from each other due to lexical scoping, but the methods are not. Remember that the flattening property ensures that the code behaves as if it were written directly inside the using class. Therefore, the *setVolume* and *setFrequency* methods have an identical implementation and thus use the same methods to obtain the upper and lower bounds. As a result, both *setVolume* and *setFrequency* try to invoke methods named “min” and “max” on a radio at runtime. But a radio does not even have methods with names “min” or “max” because a trait alias is not a true alias. Instead, an alias only inserts a delegation method that invokes the “aliased” method on the trait object. Note that even if the methods of one of one of the two traits were not aliased, the other bounded value object would always use the bound methods of the former.

Trait-based metaprogramming [19] is similar to non-conformant inheritance in Eiffel, which is discussed in the next section. Trait-based metaprogramming is discussed in more detail in the related work section.

2.4 Non-conformant Inheritance in Eiffel

Eiffel [23] and SmartEiffel 2.2 [6] support non-conformant inheritance – inheritance without subtyping – to insert code into a class through the *insert* relation. As with the regular subclassing relation, multiple inheritance and repeated inheritance (inheriting from the same class more than once) are allowed.

Fig. 4 shows how class *RADIO* implements its volume and frequency functionality using repeated non-conformant inheritance. Constructors are omitted to save space. The methods and fields of *BOUNDED_VALUE* are duplicated by giving them distinct names using renaming. In case of repeated inheritance, self invocations in Eiffel are bound *within the inheritance relation of the current method*. Suppose that the constructor of *BOUNDED_VALUE* (*make*) invokes *set_value*. When *make_vol* is executed in the constructor of *RADIO*, the *set_value* call is bound to *set_vol*, which sets the *vol* field. This is exactly the behavior that we need.

```
class RADIO
  inherit {NONE}
    -- All members must be separated explicitly.
    BOUNDED_VALUE
      rename make as make_freq,
             set_value as set_freq, value as freq,
             max as max_freq, min as min_freq end
    BOUNDED_VALUE
      rename make as make_vol,
             set_value as set_vol, value as vol,
             max as max_vol, min as min_vol end
end
class BOUNDED_VALUE[T <: COMPARABLE[T]]
  min, value, max: T
  set_value(val: T) do
    if min <= val and val <= max then value := val
end end end
```

Fig. 4. Implementing *Radio* in Eiffel

By default, members that are inherited via multiple inheritance paths form a single definition. Therefore, if we want to use the insert relation to create *Radio*, all members of the volume and frequency must be renamed individually in order to duplicate them. This not only requires a lot of work, but is also error-prone because no compile error is reported when some members are forgotten. For example, if *max* is not duplicated, the volume and frequency share the same upper bound. Another problem is that it is not possible to use two *Slider* objects for the user interface because neither the volume, nor the frequency can be used as *BOUNDED_VALUE* objects.

2.5 Manual Delegation

The *Radio* class can also be built using two *BoundedValue* objects and manually writing the delegation code, as shown in Fig. 5. Both bounded values are properly separated, but writing the delegation code is cumbersome and error-prone. Furthermore, it requires anticipation: at least one special constructor is needed in each class to allow subclasses to change the behavior of the bounded values. In the example, class *EventRadio* uses an *EventBoundedValue* to send events when the volume is changed. If the special constructor is forgotten, neither the volume nor the frequency can be customized in subclasses.

By exposing the *BoundedValue* objects via methods *volume()* and *frequency()*, the user interface for the radio can be made by connecting two *Slider* objects to the *BoundedValue* objects for the volume and the frequency.

```
class Radio {
  BoundedValue<Int> _v;
  BoundedValue<Int> volume() { return _v;}
  final Int getVolume() {return _v.getVal();}
  final void setVolume(Int v) {_v.setVal(v);}
  BoundedValue<Float> _f;
  BoundedValue<Float> frequency() { return _f;}
  final Float getFreq() {return _f.getVal();}
  final void setFreq(Float f) {_f.setVal(f);}

  Radio(Int v, Float f) {this(null,volume,null,f)}
  // The BoundedValue objects must be changable.
  Radio(BoundedValue<Int> subV, Int v,
        BoundedValue<Float> subF, Float f){
    if (subV != null) _v = subV;
    else _v=new BoundedValue<Int>(0,v,11);
    if (subF != null) _f = subFrq;
    else _f=new BoundedValue<Float>(87.5,f,108);
  }
}
class EventRadio {
  EventRadio(Int v)
    {super(new EventBoundedValue<Int>(0,v,11);}
  EventRadio(EventBoundedValue<Int> subV, Int v
        EventBoundedValue<Float> subF, Float f)
    {super(subV,v,subF,f);}
}
```

Fig. 5. Manual delegation in Java

2.6 Scala Objects

On the surface it appears that object declarations in Scala can be used. Consider for example the code in Fig. 6. The *BoundedValue* value objects for the volume and frequency are completely separated, and delegation methods are defined. A subclass *EventRadio*, however, cannot change the objects to *EventBoundedValue* objects since objects cannot be overridden in Scala. While the delegation methods can be overridden, that does not affect the behavior of the *volume* or *frequency* objects.

```
class Radio {

  // objects for volume and frequency
  object volume extends BoundedValue
  object frequency extends BoundedValue

  // providing aliases for subobject methods
  def getVolume = volume.getValue
  def setVolume = volume.setValue
  def getFrequency = frequency.getValue
  def setFrequency = frequency.setValue
}

class EventRadio extends Radio {
  // does not override volume.getValue
  override def getVolume = ...

  // generates compile error
  object volume extends EventBoundedValue
}
```

Fig. 6. Implementing *Radio* with objects in Scala

2.7 Summary

Even though the radio example is simple, none of the typical reuse techniques is able to maintain that simplicity in the implementation. Mixins and traits cannot be used at all because they do not offer support for using multiple building blocks of the same type. Non-conformant inheritance allows easy customization without anticipation, but requires a lot of work to separate the building blocks, and does not allow them to be used as if they were separate objects. With manual delegation, the situation is exactly the other way around. Isolating the building blocks and using them as separate objects is easy, but customization is less elegant and requires anticipation.

3 Subobject-Oriented Programming

Subobject-oriented programming augments object-oriented programming with subobjects, which allow developers to capture cross-cutting structural boilerplate code in a class and then use it as a configurable building block to create other classes. A subobject can be seen as a combination of inheritance and delegation. It combines the ability to have isolated and accessible building blocks, as provided by delegation, with the ability to easily modify their behavior without anticipation and integrate them in the interface of the composed class, as with inheritance.

Subobject members can be exported to the surrounding class, and customized either in the subobject itself or in the surrounding context. Subobjects can be refined in subclasses of the composed class, they can be treated as real objects.

Fig. 7 shows the syntax of subobjects. Subobjects are *named* members of the composed class. Type T is the *declared superclass* of the subobject. Members defined in the body of a subobject may also redefine members of T. Within the body of a subobject, the *this* expression refers to the subobject itself. The *outer* expression refers to the (sub)object that directly encloses the subobject. The value of *outer* is equal to the value of *this* in the enclosing (sub)object. Similar to invocations on *this*, invocations on *outer* are bound dynamically. Section 3.2 explains *export* clauses. Section 3.3 explains subobject refinement, *overrides* and *refines* clauses, and *super* calls. Section 3.4 explains subobject initialization. Note that we use the Scala syntax for methods throughout the paper since it is more concise than the syntax of Java.

Class ::= **`class`** *T* **`extends`** *T* $\overline{\textbf{\texttt{implements}}\ T}\{\overline{Member}\}$
Member ::= *Method* | *Field* | *Subobject* | *Export*
Subobject ::= **`subobject`** *Id T [Body]*
Export ::= **`export`** $\overline{Path\ [\textbf{\texttt{as}}\ Id]}$
Refines ::= *Id* **`refines`** *Path*
Overrides ::= *Id* **`overrides`** *Path*
Path ::= $\overline{Id}$
SubobjectInit ::= **`subobject`**.*Path*($\overline{e}$)
SuperCall ::= *[Prefix .]* **`super`** *[. Path] .m*($\overline{e}$)
Prefix ::= $\overline{\textbf{\texttt{outer}}}$ | *Path* | $\overline{\textbf{\texttt{outer}}}$.*Path*

Fig. 7. Syntax for subobjects

3.1 Subobject Basics

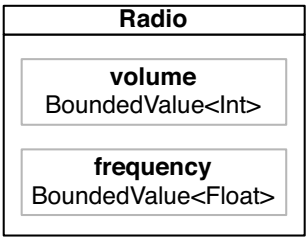
Fig. 8 shows how *Radio* implements its volume and frequency with subobjects named *volume* and *frequency*. These subobjects have declared superclasses *BoundedValue<Integer>* and *BoundedValue<Float>* respectively. The interface of *Radio* does not yet contain the setters and getters for the volume and the frequency.

```
class Radio {
  subobject volume BoundedValue<Integer>;
  subobject frequency BoundedValue<Float>;
}

class BoundedValue<T> {
  BoundedValue(T min, T val, T max) {...}

  T getValue() = value;
  void setValue(T value) {
    if(value >= getMin() && value <= getMax())
      {this.value = value;}
  }
  T value;

  // similar code for the min and max values
}
```

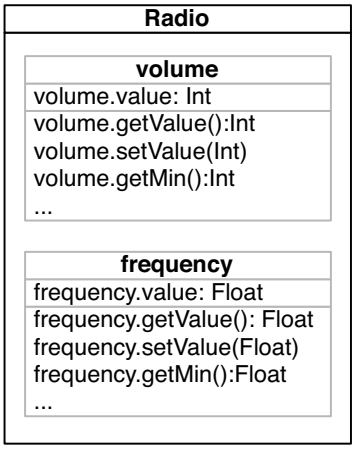
Fig. 8. Using subobjects for the volume and the frequency

The diagram in Fig. 9a shows the class diagram for *Radio*. A subobject is visualized as a box inside the composed class because it is used as a building block for that class. The declared super class of the subobject is shown underneath its name.

Subobjects are isolated by default, without requiring renaming or exporting. The diagram in Fig. 9b shows a flattened view of the members of the *Radio* class of Fig. 8. The members that are directly available in the interface of *Radio* are shown in bold.



(a) The class diagram of *Radio*



(b) A flattened view of the members of *Radio*

Fig. 9. Visualization of class *Radio*

Subobject *volume* introduces the following members into *Radio*: *volume.getValue*, *volume.setValue*, *volume.value*, *volume.getMin*, and so forth. Similarly, subobject *frequency* introduces *frequency.getValue*, *frequency.setValue*, and so forth. This avoids an explosion of name conflicts in the enclosing class.

Within the context of a subobject, *this* binds to that subobject. Therefore, method calls and field accesses executed in the context of a subobject are bound within that subobject. For example, to verify that the value of a *BoundedValue* is not set to an invalid value, the *setValue* method of class *BoundedValue* must call *getMin* and *getMax* to obtain the bounds. During the execution of *volume.setValue*, these calls are bound to *volume.getMin* and *volume.getMax* respectively.

Using Subobjects as Real Objects. The name of a subobject allows it to be used as a real object whose type is a subtype of its declared superclass. The subtype reflects any customization done in the subobject body, such as using a more specific return type. Fig. 10 illustrates how a subobject is used as a real object. Invoking *r.volume* returns an object of type *Radio.volume*, which is a subtype of *BoundedValue<Integer>*. This allows the subobject to be connected to a slider of the user interface.

```
class Slider<T extends Number> {
  BoundedValue<T> _model;
  void connect(BoundedValue<T> bv) {...}
}
Radio r = new Radio();
Slider<Int> vs = new Slider<Int>();
vs.connect(r.volume); // similar for frequency
```

Fig. 10. Using subobjects as real objects

Nested Subobjects. Nested subobjects are also available in the composed class. The code in Fig. 11 shows a part of the radio example with nested subobjects. Note that this version is incomplete as it does not perform any bounds checks. The example merely serves to illustrates nesting. In Sect. 3.2 we show how subobject members can be made available directly in the interface of the composed class. In Sect. 3.3 we show how subobject members can be overridden to enforce the upper and lower bounds of *BoundedValue*. Class *BoundedValue* now uses three *Property* subobjects with names *value*, *min*, and *max* for the value and the bounds. Class *Property* has methods *getValue* and *setValue* and field *value*. In this case, *BoundedValue* does not offer *getValue* and *setValue* directly in its interface.

The diagram in Fig. 12 shows the members of *Radio* in this scenario. Class *Radio* now has members *volume.value.getValue*, *volume.min.getValue*, *volume.max.getValue*, and so forth. Note that a developer is not confronted with a large number of members when looking at the interface of *Radio*. Only the members shown in bold are directly accessible. The other can only be accessed via the subobjects.

```
class Radio {
  subobject volume BoundedValue<Integer>;
  subobject frequency BoundedValue<Float>;
}

class BoundedValue<T> {
  subobject min Property<T>;
  subobject value Property<T>;
  subobject max Property<T>;
}

class Property<T> {
  T value;
  T getValue() = value;
  void setValue(T t) {value = t;}
}
```

Fig. 11. Part of the radio example with nested subobjects

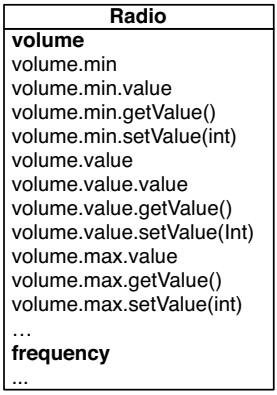


Fig. 12. A flattened view of the members of *Radio* (nested version)

3.2 Exporting Subobject Members

While a subobject can be accessed as an object, it is more convenient if commonly used members are available directly in the composed class. In addition, it is desirable to give such members names that are appropriate for the composed class. Subobject members are added to the interface of the composed class using *export* clauses. In Fig. 13, the getter and setter methods for the volume and the frequency *BoundedValue*s are exported to *Radio* as *getVolume*, *setVolume*, *getFrequency*, and *setFrequency*. Whether the getter and setter methods are implemented directly in *BoundedValue* or exported to *BoundedValue* from nested subobjects makes no difference for class *Radio*.

```
class Radio {
  subobject volume BoundedValue<Int> {
    export getValue as getVolume,
           setValue as setVolume;
  }
  subobject frequency BoundedValue<Float> {
    export getValue as getFrequency,
           setValue as setFrequency;
  }
}
```

Fig. 13. Adding getters and setters to *Radio*

The class diagram of *Radio* with the exported members is shown in Fig. 14. The getter and setter methods of the *volume* subobject can not be invoked directly on an object of type *Radio* via *getVolume* and *setVolume* respectively. The notation $m < f$ indicates that member f from the subobject is exported to the composed class as m. The new name is written at the left side to improve the readability of the interface of the composed class.

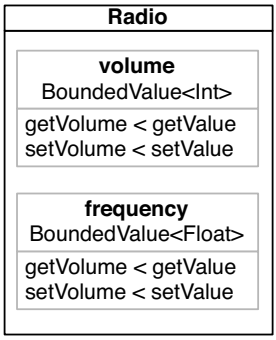


Fig. 14. The class diagram of *Radio* with exported members

An export clause ***export*** `path.d` in subobject *s* makes member *s.path.d* accessible via name *d* in the enclosing scope, so long as doing so does not create name conflicts. The form ***export*** `path.d` ***as*** `Id` can also be used to give a new name to the exported path. In both cases, the member is still available via its original path (`s.path.d`).

The alias defined by an export clause cannot be broken, as shown in Fig. 15. Class *BrokenRadio* overrides *setVolume* to set the value to the opposite value of the scale. Because of the aliasing, *volume.setValue* is also overridden when *setVolume* is overridden. Thus regardless of whether the client changes the volume via *setVolume* or via *volume.setValue*, the effect is always the same.

```
class BrokenRadio extends Radio {
  void setVolume(Int vol) {super.setValue(11-vol);}
}
BrokenRadio br = new BrokenRadio();
br.setVolume(1); // write directly
//read through subobject equals direct read
assert(br.volume.getValue() == br.getVolume());
br.volume.setValue(2); //write via subobject.
//read through subobject equals direct read
assert(br.volume.getValue() == br.getVolume());
```

Fig. 15. Overriding cannot break aliases

Export clauses provide the best of two worlds: ease of use and reuse. The composed class can provide an intuitive and uncluttered interface. Classes meant to be used as subobjects can provide a lot of functionality without cluttering the composed classes. Members that are not exported can still be accessed by using the subobject as a separate object whose type is its declared superclass, as illustrated in Fig. 16. In other approaches, such members are no longer available, resulting in code duplication.

3.3 Customizing Subobjects

Methods of a subobject can be overridden in its body. The return type is covariant and the parameter types are invariant. Otherwise the code in the subobject may break.

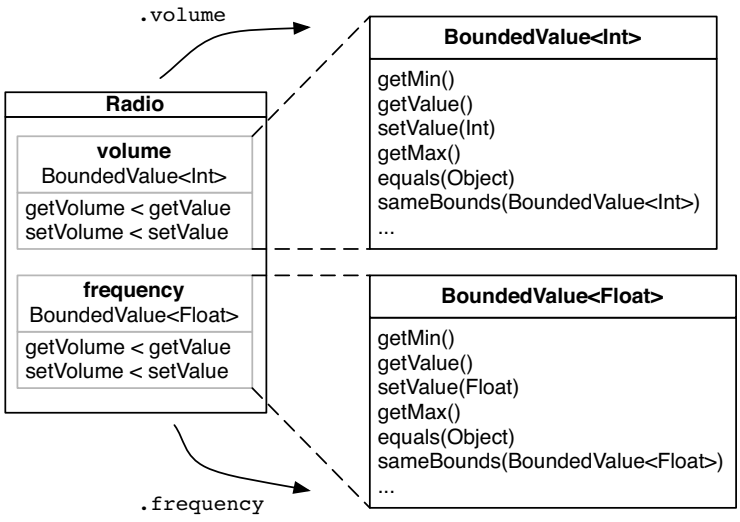


Fig. 16. Zooming in on a subobject

```
class BoundedValue<T extends Comparable<T>> {
 subobject value Property<T> {
   export getValue, setValue;
   boolean isValid(T t) =
     (t != null &&
     outer.getMin() <= t && t <= outer.getMax())
 }

 subobject min Property<T> {
   export getValue as getMin, setValue as setMin;
   boolean isValid(T t) =
     (t!= null && t <= outer.getValue())
 }

 subobject max Property<T> {
   export getValue as getMax, setValue as setMax;
   boolean isValid(T t) =
     (t!= null && outer.getValue() <= t)
 }
}
```

Fig. 17. Overriding subobject members

Suppose that *BoundedValue* is implemented using three *Property* subobjects for the value and the bounds, as shown in Fig. 17. The setter of *Property* uses *isValid* to validate the given value. Class *BoundedValue* redefines the *isValid* methods of the three subobjects to ensure that the value will be between the upper and lower bounds. The subobjects are isolated by default, thus *outer* and *export* are used to cross the boundaries of the subobjects.

Fig. 18 shows the class diagram of *BoundedValue*. The members that are overridden in the subobject are shown in a separate area.

The diagram in Fig. 19 shows the lookup table of *Property* and a part of the lookup table of *BoundedValue*. For each subobject, there is an additional lookup table. Field reads and writes are bound dynamically. Note that the lookup table for a subobject contains a new entry for each field of the declared superclass. A new entry is needed because the position of the fields of a subobject in the memory layout of the object of the outer class is specific for each outer class. The *getValue* and *setValue* methods of *BoundedValue.value* point to the corresponding implementations in *BoundedValue*, while its *isValid* method uses the overriding implementation M_x.

Overriding in an Enclosing Scope. In some cases, the overriding method cannot be written directly in the subobject, for example, when a programmer wants to override a method of a subobject using a method the outer class inherits from a superclass. Another example is when methods of different subobjects need to be joined to share the same implementation. In these situations, an *overrides* clause written in an outer scope can be used to achieve the desired effect.

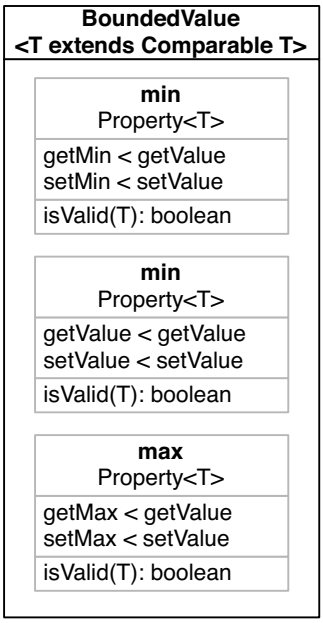


Fig. 18. The class diagram of *BoundedValue*

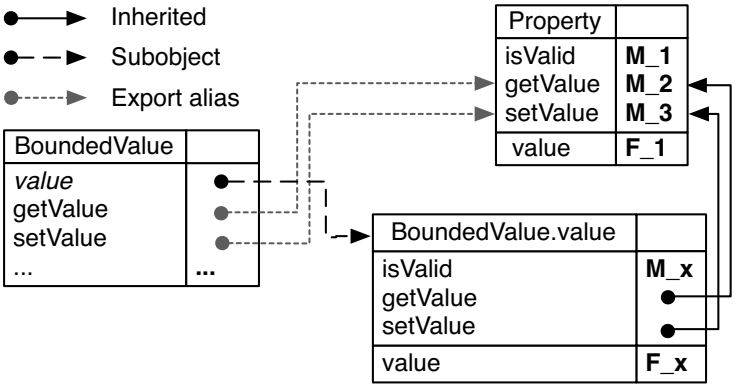


Fig. 19. Part of the lookup tables of *BoundedValue*

```
class StereoRadio {
  subobject frequency BoundedValue<Float> {
    export getValue as getFrequency,
           setValue as setFrequency;
  }
  subobject left BoundedValue<Int> {
    export getValue as getLeftVol,
           setValue as setLeftVol;
  }
  subobject right BoundedValue<Int> {
    export getValue as getRightVol,
           setValue as setRightVol;
  }
  setMaxVolume overrides left.setMax;
  setMaxVolume overrides right.setMax;
  void setMaxVolume(Int v) {
    left.super.setMax(v);
    right.super.setMax(v);
  }
  isValidMaxVolume overrides left.max.isValid;
  isValidMaxVolume overrides right.max.isValid;
  void isValidMaxVolume(Int v) =
    left.max.super.isValid(v) && right.max.super.isValid(v)
}

StereoRadio r = new StereoRadio();
// Equivalent ways of setting the maximum volume
r.setMaxVolume(1);
r.left.setMax(1);
r.right.setMax(1);

// Both maximum volumes are always the same.
invariant(r.left.getMax() == r.right.getMax());
```

Fig. 20. Joining parts of two subobjects

Suppose that we need a stereo radio for which the minimum and maximum volume of the left and the right channel are always the same. The code in Fig. 20 shows how this is implemented by joining the *BoundedValue* subobjects for both maximum volumes together; the code for the minimum volume is similar. *StereoRadio* defines a new setter for the maximum volume which invokes the overridden methods of both subobjects to set the maximum volumes. The overrides clauses specify that *setMaxVolume* overrides the *setMax* methods of both subobjects. Similar to export clauses, an overrides clause defines an alias relation between method names that cannot be broken. Therefore, the maximum volume of both channels is changed regardless of whether it is changed via *setMaxVolume* or via one of the subobjects. Without the ability to override methods in

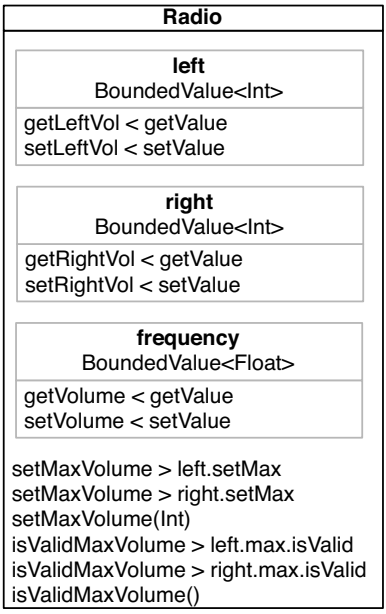


Fig. 21. The class diagram of *StereoRadio*

the outer scope, these methods would have to be overridden in the subobjects, resulting in code duplication. The diagram in Fig. 21 shows the class diagram of *StereoRadio*. The $>$ symbol represents an override clause that declares that the left-hand side overrides the right-hand side.

Merging Fields. The overrides clause can also be used to join fields of nested subobjects. For example the bounds of left and right volume of the stereo radio can also be joined by overriding *isValid* as in Fig. 20 or by merging the fields of the bounds instead. This alternative is illustrated in Fig. 22.

```
class StereoRadio {
  maxVolume overrides left.max.value;
  maxVolume overrides right.max.value;
  Integer maxVolume;
}
```

Fig. 22. Merging fields of two subobjects

Subobject Refinement. As subobjects are class members, they can also be modified in a subclass. Contrary to methods, subobjects are not overridden but are instead *refined*. This is similar to refinement (or further binding) of nested classes in Beta [17] and gbeta [10]. Our previous approach [26] allowed subobjects to be completely overridden, but this was fragile and required code duplication.

```
class EventRadio extends Radio {
  subobject frequency EventBoundedValue<Float>;
}
```

Fig. 23. Changing the declared superclass of a subobject

Suppose that we want to create a subclass of *Radio* that sends events when the bounds or the value of the frequency are changed. Fig. 23 shows how this can be implemented using subobject refinement. We assume that *EventBoundedValue* is a subclass of *BoundedValue* that sends events. Class *EventRadio* refines the *frequency* subobject of *Radio* by changing its declared superclass to *EventBoundedValue*. The export clauses are inherited from *Radio.frequency*.

When subobject t refines subobject s, all members of s are inherited by t. A member x defined in the body of t overrides or refines a member of s, depending on whether x is a subobject or not. For export clauses, the existing name mapping cannot be changed; only additional mappings can be added. The declared superclass of t is equal to the declared superclass of s, unless t specifies its own declared superclass (T), in which case T must be a subtype of S.

Suppose now that we want to send an event only when the actual volume changes. In this case, we refine only the nested *value* subobject of the *volume* subobject. This is shown in Fig. 24. The *frequency* subobject inherits its declared superclass (*BoundedValue*<*Float*>) from the *frequency* subobject of *Radio*. Assume that *EventProperty* is a subclass of *Property* that sends events. The diagram in Fig. 25 shows how nested refinement is visualized in a class diagram. The subtype relations between the types of the subobjects involved in the refinement are not shown because that would clutter the diagram.

```
class EventRadio extends Radio {
  subobject frequency {
    subobject value EventProperty<Float>;
  }
}
```

Fig. 24. Nested subobject refinement

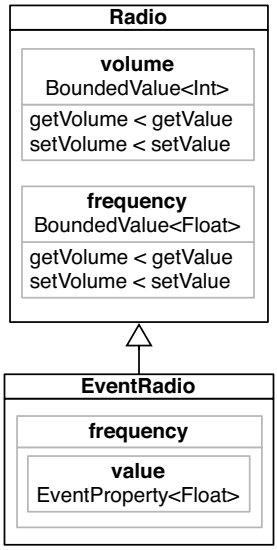


Fig. 25. A class diagram of nested refinement

Super Calls. Subobject refinement gives rise to a form of multiple inheritance because a subobject inherits members from the refined subobject and possibly a new declared superclass. Conflicts are detected using the *rule of dominance*, as used in C++ and Eiffel. If a subobject would inherit two different definitions for a member, then it must provide a new definition to resolve the conflict.

Suppose for example that *Radio.frequency* overrides *setValue* from its declared superclass *BoundedValue* to check whether the current frequency matches a pre-programmed channel. Class *EventBoundedValue* also overrides *setValue* to send events, which means that *EventRadio.frequency* has two candidate *setValue* methods. This conflict is resolved by defining a new *setValue* method as shown in Fig. 26.

Overriding methods typically use super calls to access overridden implementations. Because of the multiple inheritance, super calls must be disambiguated. Super calls therefore have the form `prefix.super.suffix.m(args)`. The semantics of a super call is as follows. The suffix and the method call are looked up in the scope determined by the prefix. If the prefix ends with a path (*Path* or $\overline{\texttt{outer}}$*.Path*), then the scope is that subobject. If the prefix is empty or $\overline{\texttt{outer}}$, the scope is the superclass of the enclosing scope referenced by the prefix. For a subobject, the superclass is the declared superclass. All binding in a super call is static.

In Fig. 26, expression `super.setValue(val)` invokes the *setValue* method of declared superclass *EventBoundedValue*, and `outer.super.frequency.setValue(val)` invokes the *setValue* method of *Radio.frequency*. In the super call, **outer** jumps to the enclosing class (*EventRadio*), **super** jumps up to *Radio*, and finally *frequency* jumps inward to the *frequency* subobject. In this case, the value is set twice, but we chose this implementation to illustrate how to access a refined subobject with a super call.

```
class Radio {
  subobject frequency BoundedValue<Float> {
    void setValue(Float val) {
      super.setValue(val);
      checkForPreProgrammedStation();}
  }
}
class EventRadio extends Radio {
  // refinement changes declared superclass
  subobject frequency EventBoundedValue<Float> {
    void setValue(Float val) {
      super.setValue(val);
      outer.super.frequency.setValue(val);
  }
}
```

Fig. 26. Accessing overriding methods

Note that if *setValue* was overridden in *Radio* instead of in *Radio.frequency*, there would still be a conflict. Even though the *setValue* method would then not lexically be in the *frequency* subobject, it would still be the *setValue* method of *frequency* and thus cause a conflict with *EventBoundedValue.setValue*.

Refining Subobjects in an Enclosing Scope. Similar to methods and fields, a subobject can also be redefined in a scope that encloses the subobject. A redefining subobject in an outer scope inherits all regular members from all redefined subobjects, but export clauses are not inherited. All redefined subobjects are joined into a single subobject.

We illustrate subobject refinement in an outer scope for the stereo radio example from Section 3.3. Remember that the bounds of both volume channels should always be the same. The previous approach (Fig. 20) joined the *setMax* methods from the *left* and *right* subobjects. This is unwieldy if many methods need to be joined. An alternative approach is to join the nested *Property* subobjects that represent the bounds. Fig. 27 shows how the nested subobjects for the bounds of the volumes can be joined. The *refines* clauses join the nested *max* subobjects of both channels.

The example in Fig. 27 illustrates the need to customize the rule of dominance for subobjects. Remember from Fig. 17 that *BoundedValue* overrides the *isValid* methods of its three *Property* subobjects to do the bounds check. As such, the *isValid* method of the *max* subobject of *BoundedValue* is more specific than *Property.isValid*. Using the traditional rule of dominance, no conflict would be reported because the *isValid* methods from both *left.max* and *right.max* originate from the same definition in *BoundedValue*. The behavior of these methods, however, is not at all the same. Invoking *left.max.isValid(val)* checks whether *val* is not smaller than the left volume, whereas *right.max.isValid(val)* checks whether *val* is not smaller than the right volume. This means that there are actually two most specific candidates instead of one. Therefore, both methods are overridden by a unique most specific version.

```
class Radio {
  subobject left BoundedValue<Int> {
    export getValue as getLeftVol,
           setValue as setLeftVol;
  }
  subobject right BoundedValue<Int> {
    export getValue as getRightVol,
           setValue as setRightVol;
  }

  // join both upper bounds
  maxVolume refines left.max;
  maxVolume refines right.max;
  subobject maxVolume Property<Int> {
    // This method must be redefined because
    // the original definition captures its
    // context.
    boolean isValid(Int val) =
      (outer.left.max.super.isValid(val) &&
       outer.right.max.super.isValid(val))
  }
  // similar for the minimum volume
}

Radio r = new Radio();
// equivalent ways of setting the max volume
r.maxVolume.setValue(1);
r.left.setMax(1);
r.right.setMax(1);

// both channels always have the same max value
invariant(r.left.getMax() == r.right.getMax());
```

Fig. 27. Refining nested subobjects in the composed class

More formally, when member m is redefined in a subobject or in an enclosing scope, the redefinition of m can access all elements of the composed class T that contains the subobject. Therefore, the new definition of m depends on T. Suppose that subobject s refines multiple nested subobjects $\overline{x}.t$ of type T. Each member m_i of nested subobject $x_i.t$ depends on x_i and thus has a behavior that is potentially different from all m_j with $i \neq j$. As a result, no m_i can be selected automatically as *the* version of m. Therefore all member m_i conflict with each other in the context of s.

3.4 Initialization of Subobjects

During the construction of an object, its subobjects must be initialized as well. Initialization of a subobject is similar to a traditional super constructor call. No additional object

is created, but the initialization code is executed on the new object of the inheriting class. In case of a subobject initialization call, however, the initialization code is executed on the *part* of the new object of the composed class that corresponds to the subobject. Syntactically, a subobject constructor call consists of the keyword ***subobject*** followed by the name of the subobject and the arguments passed to the constructor. Subobject constructors must be invoked directly after the super constructor calls. If the class of a subobject has a default constructor, no explicit subobject constructor call is required for that subobject. A subobject inherits all constructors from its declared superclass, but it cannot define constructors itself.

Consider the example in Fig. 28. To initialize its subobjects, the constructor of *Radio* performs two *subobject constructor calls*. Both calls invoke the same constructor of *BoundedValue*, but each call operates on a different part of the *Radio* object. Similarly, the constructor of *BoundedValue* invokes the constructor of *Property* for each of its three subobjects.

```
class Radio {
  Radio(Integer vol, Float freq) {
    // initialize the volume subobject
    subobject.volume(0,vol,11);
    // initialize the frequency subobject
    subobject.frequency(87.5,freq,108);
  }

  subobject volume BoundedValue<Integer> {...}
  subobject frequency BoundedValue<Float> {...}
}
```

Fig. 28. Initializing the subobjects of a radio

Initialization of Refined Subobjects. If a subobject is refined, initialization is more complicated. Whether the original subobject constructor calls remain valid depends on whether the declared superclass of the subobject has changed.

```
class TeenagerRadio extends Radio {
  TeenagerRadio(Float freq) {super(0,freq);}
  subobject volume {Float getValue() = 11}
}
```

Fig. 29. The refined subobject does not change the declared superclass

We explore the different scenarios using a special class of radios for teenagers, whose volume is always set to the maximum. The first way to implement *TeenagerRadio* is to override the getter for the volume by refining the *volume* subobject and overriding *getValue*, as shown in Fig. 29. In this case, *TeenagerRadio.volume* is an anonymous

subclass of *Radio.volume*, and thus the former inherits its constructors from the latter. This is similar to constructors of anonymous inner classes in Java. As a result, the subobject constructor call in *Radio* remains valid for *TeenagerRadio.volume*.

The second way to create the teenager radio is to change the declared superclass of the *volume* subobject, as shown in Fig. 30. Suppose that *MaxBoundedValue* is a subclass of *BoundedValue* in which *getValue* always returns the upper bound. Because the *volume* subobject now has a different declared superclass, the subobject constructor call for *volume* in *Radio* is no longer valid. Therefore, the constructor of *TeenagerRadio* must perform the subobject constructor call itself.

```
class TeenagerRadio extends Radio {
  TeenagerRadio(Float freq) {
    super(0,freq);
    subobject.volume(0,11);
  }
  subobject volume MaxBoundedValue<Integer>;
}
```

Fig. 30. The declared superclass is changed

The subobject constructor call for *volume* in *TeenagerRadio replaces* the subobject constructor call of *volume* in *Radio*. The latter is no longer executed when initializing a *TeenagerRadio*. Instead, the call in *TeenagerRadio* is executed *at the moment the subobject constructor call of* Radio.volume *would have been executed*. This ensures that the subobject is still initialized when the code following the subobject constructor call in *Radio* is executed.

Initialization of Nested Refined Subobjects. So far, we implemented *TeenagerRadio* by refining the *volume* subobject, which is a direct subobject of *Radio*. But since *BoundedValue* itself uses a *Property* subobject for its value, we can also refine the nested *value* subobject of *volume*.

The third way to implement *TeenagerRadio* is to override *getValue* in *volume.value*, as shown in Fig. 31. Because the declared superclasses of the *volume* and *volume.value* subobjects have not changed, no explicit subobject constructor calls are needed.

The fourth and final way to implement *TeenagerRadio* is to change the declared superclass of *volume.value*. In the code in Fig. 32, class *Eleven* is a subclass of *Property<Integer>* that always returns 11 as its value. The superclass of the *value* subobject of the *volume* subobject is then redefined to *Eleven*. Therefore, a new subobject constructor call is required to initialize the *volume.value* subobject. In this specific case the subobject constructor call is optional because *Eleven* has a default constructor.

The subobject constructor call for a nested subobject must be written in a constructor of the *outermost* class to avoid ambiguities. Otherwise, a constructor definition would have to be written inside the enclosing subobject body. But this could lead to the typical

```
class TeenagerRadio extends Radio {
  TeenagerRadio(Float freq) {super(0,freq);}
  subobject volume {
    subobject value {
      Float getValue() = 11
    }
  }
}
```

Fig. 31. Nested refinement without changing the declared superclass

```
class TeenagerRadio extends Radio {
  TeenagerRadio(Float freq) {
    super(0,freq);
    // Optional: Eleven has a default constructor.
    subobject.volume.value();
  }
  subobject volume {subobject value Eleven;}
}
```

Fig. 32. Nested refinement changes the declared superclass

problems with initialization in a multiple inheritance hierarchy when that subobject is refined. Therefore, subobjects cannot contain constructors. If the host language already supports multiple inheritance, this could be allowed, but we do not want to force this problem onto a host language with single inheritance.

If a subobject redefines multiple subobjects, an explicit subobject constructor call is required. That call is executed *the first time* one of the redefined subobjects would been initialized. Since there is always one most specific version of a subobject, there is always one subobject constructor call used to initialize the subobject, namely, the subobject constructor call that corresponds to the most specific version.

A remaining issue with subobject initialization is that the superclass constructor can rely on properties of the subobjects after they have been initialized. Therefore, there is a need to be able to specify these properties. If a subclass explicitly initializes a subobject, it must then ensure that these properties hold after the initialization of that subobject. A mechanism to define such contracts is a topic for future work.

4 Illustrations of Subobject-Oriented Programming

The radio example suffices to illustrate how subobject-oriented programming works, but it is deceptively simple. In this section, we illustrate the possibilities of subobject-oriented programming using classes in the JLo library. Section 4.1 presents the classes for defining associations. Section 4.2 shows how to reuse advanced graph algorithms by building graphs on top of associations.

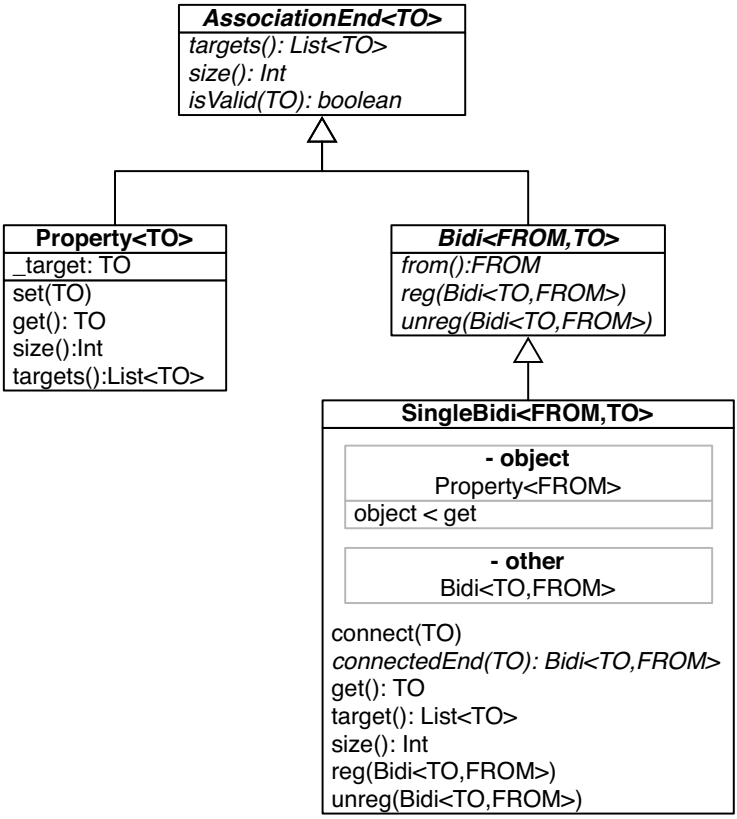


Fig. 33. A partial class diagram for the association classes

4.1 Subobjects for Associations

The library of JLo, the subobject-oriented extension of Java, contains classes for uni- and bi-directional associations. A subobject is used for each navigable end of an association. The code in Fig. 33 shows a class diagram of the association classes and their most important methods. The corresponding code is shown in Fig 34. Most definitions are omitted, and some names have been abbreviated to save space. The association classes in the JLo library also use wildcards in the type arguments to increase the flexibility. Wildcards are omitted as they are not needed to illustrate the use of association subobjects. The top interface provides only the functionality to query an association. Class *Property* represents an encapsulated field, which is a unidirectional association. Similar classes are defined for sets and lists.

A bidirectional association end is connected to the object on its side of the association, and offers methods for registering and unregistering other bidirectional association ends. The *reg* and *unreg* methods keep the association in a consistent state, and make it possible to mix unary and n-ary association ends. The *reg* method of a unary end

```
interface AssociationEnd<TO> {
  List<TO> targets();
  int size();
  boolean isValid(TO t);
}

class Property<TO> implements AssociationEnd<TO> {
  TO _target
  int size() = 1;
  void set(TO t) {if(isValid(t) {_target = t}}
  TO get() = _target
  List<TO> targets() = List(get())
}

interface Bidi<FROM,TO> extends AssociationEnd<TO> {
  FROM object();
  // internal bookkeeping methods
  void reg(Bidi<TO,FROM> b);
  void unreg(Bidi<TO,FROM> b);
}

abstract class SingleBidi<FROM,TO> implements Bidi<FROM,TO> {

  private subobject object Property<FROM> {
    export get as object;
  }
  private subobject other
           Property<Bidi<TO,FROM>>;

  TO get() = other.get().object()
  List<TO> targets() = List(get())
  int size() = 1;
  void connect(TO t) = {
    Bidi<TO,FROM> b = connectedEnd(t);
    reg(b);
    if (b != null) {b.reg(this.other);}
  }
  void reg(Bidi<TO,FROM> o) = {... other.set(o) ... }
  void unreg(Bidi<TO,FROM> o) = {... other.set(null) ... }
  abstract Bidi<TO,FROM> connectedEnd(TO t);
}
```

Fig. 34. Classes for association ends

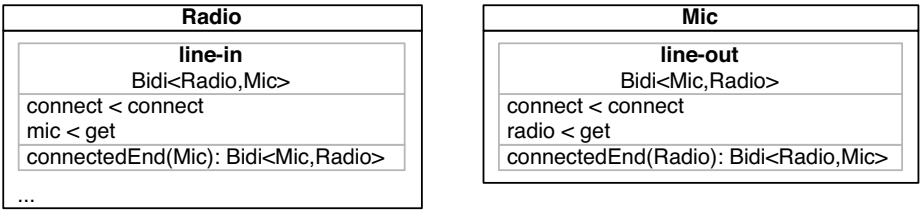


Fig. 35. Connecting a microphone to the radio

```
class Radio {
  ...
  subobject line-in SingleBidi<Radio,Mic> {
    export connect, get as mic;
    Bidi<Mic,Radio> connectedEnd(Mic m) = m.line-out;
  }
}

class Mic {
  ...
  subobject line-out SingleBidi<Mic,Radio> {
    export connect, get as radio;
    Bidi<Radio,Mic> connectedEnd(Radio r) = r.line-in;
  }
}

Radio r = new Radio();
Mic m1 = new Mic();
Mic m2 = new Mic();
r.connect(m1);
assert(r.mic() == m1 && m1.radio() == r);
r.connect(m2);
assert(r.mic() == m2 && m2.radio() == r);
assert(m1.radio() == null);
```

Fig. 36. Connecting a radio to a microphone

disconnects from its current connection (if any) using *unreg* and connects to the given association end. The *reg* method of an n-ary association end simply adds the given association end.

Abstract class *SingleBidi* represents unary bidirectional associations. The *Property* subobjects to store the object at its own end and the connected association end are private to hide their setters. The exported methods are still public. The *connect* and *disconnect* methods uses the *reg* and *unreg* methods to keep the association consistent. The abstract method *connectedEnd* determines the subobject at the other end of the association. It is implemented in the actual subobjects in the application. Similar classes are

defined for n-ary bidirectional associations with set and list semantics. The library also provide classes for passive bidirectional associations, which do not provide a *connect* method because they are connected to different subobjects of different classes.

The class diagram in Fig. 35 and the code in Fig. 36 show how we can connect a microphone to a radio. The bidirectional association is implemented by simply specifying the association ends that must be connected. This is much simpler than writing the logic for keeping the association in a consistent state. Many programmers forget to clean up the back-pointers on at least one side of the association.

4.2 Subobjects for Graphs

The JLo library also contains classes to build weighted and unweighted graphs on top of the associations. The classes in the library allow advanced graph layouts, but for reasons of space we only present simple classes for homogeneous graphs.

```
abstract class DigraphNode<V> {
  abstract List<V> successors();
  abstract DigraphNode<V> node(V v);
  boolean isPredecessorOf(V v) = {...}
  List<V> allSuccessors() = {...}
  ...
}
```

Fig. 37. A class for graph nodes

The code in Fig. 37 shows the top class of the graph library. The abstract *edges* method of *DigraphNode* must return the direct successor objects. The *node* method determines to which graph node of the direct successor this graph is connected. Based on this method, basic graph functionality can be implemented such as computing all successors of the current node.

The class diagram in Fig. 38 and the code in Fig. 39 show how graphs can be defined. A *Klass* has subobjects for its name, a list of superclasses, and a list of subobjects. A *Subobject* has subobjects for its name and its *Klass* (the declared superclass). A transitive association is used to define an association from a *Klass* to the declared superclasses of its subobjects. These associations are then used to build a type graph and an inheritance graph. The *isValid* methods of both *ListProperty* subobjects are overridden to forbid loops in the inheritance graph. Only simple configuration code was written, to obtain full graph functionality.

An alternative implementation could used graph nodes that support *AssociationEnd* subobjects as edges. In that case, *typeGraph* would be a graph node subobject that uses the *superKlasses* as its source of edges. For the inheritance graph, a *TransitiveAssociationEnd* subobject would be used and configured to use the *Klass.subobjects* and *Subobject.klass* subobjects. This transitive association subobject would then be used as a source of edges for the inheritance graph, together with *superKlasses*.

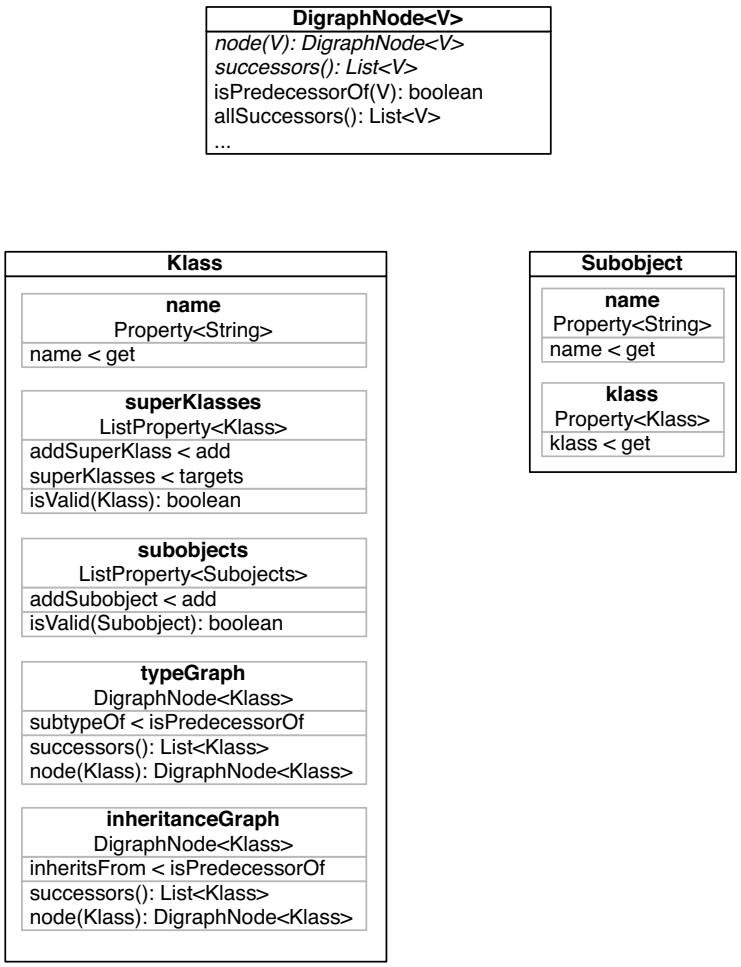


Fig. 38. A class diagram of class with graph subobjects

```
class Klass {
  subobject name Property<String> {...}
  subobject superKlasses ListProperty<Klass> {
    export add as addSuperKlass,
         targets as superKlasses;
    boolean isValid(Klass klass) = !klass.inheritsFrom(outer);
  }
  subobject subobjects ListProperty<Subobject> {
    export add as addSubobject;
    boolean isValid(Subobject s) =
      !s.getKlass().inheritsFrom(outer);
  }

  // define graphs on top of the associations
  subobject typeGraph DigraphNode<Klass> {
    export isPredecessorOf as subtypeOf
    List<Klass> successors() = {
      // collect Klasses referenced by the subobjects
      ... subobject.getKlass() ...
    }
    DigraphNode<Klass> node(Klass klass) = klass.typeGraph;
  }

  subobject inheritanceGraph DigraphNode<Klass> {
    export predecessorOf as inheritsFrom
    List<Klass> successors() = {
      // collect Klasses referenced by the subobjects
      ... subobject.getKlass() ...
      // add the superklasses
      ... outer.superKlasses() ...
    }
    DigraphNode<Klass> node(Klass klass) = klass.inheritanceGraph;
  }
}
class Subobject {
  subobject name Property<String> {...}
  subobject klass Property<Klass> {...}
}
```

Fig. 39. Adding graph functionality with subobjects

Weighted Graphs. The JLo library also defines classes for weighted graphs, as shown in Fig. 40. Weighted graph node use explicit edges because each edge has its own weight. Class *WeightedNode* restricts the edges to weighted association ends such that it can offer additional functionality for weighted graphs. The *WeightedEnd* class represents a weighted association to objects of type *V* via intermediate objects of type *E*. The *first* and *second* methods return the *V* objects connected by the intermediate object such that *otherEnd* can compute the target objects of the association.

```
abstract class WeightedNode<V> implements DigraphNodeWithEdge<V> {
  abstract List<WeightedEnd<V,?>> edges();
  abstract WeightedNode<V> node(V v);
  Double shortestDistanceTo(V v) = {...}
  ...
}
abstract class WeightedEnd<V,I> implements AssociationEnd<V> {
  abstract List<I> intermediates();
  abstract Double weight(I i);
  abstract V first(I i);
  abstract V second(I i);
  V otherEnd(V v) =
     if(first(object()) == v) second(object())
     else first(object())
  ...
}
```

Fig. 40. Library classes for weighted graphs

The use of the graph subobjects is illustrated in Fig. 41. A road has a length and is connected to two cities via bidirectional associations. The association ends in *Road* are implemented with subobjects *first* and *second*. Both are connected to *City.roads* by implementing *connectedEnd*.

Subobject *City.cityToCity* represents the weighted edge between two cities. Subobject *City.roadNetwork* implements the abstract methods of *WeightedNode* to select the edges and to select the graph node of the connected cities. It also exports the method to compute the shortest path to another city.

In a standard object-oriented style, where the graph structure is implemented with lots of low-level fields and methods, the graph algorithms would typically be reimplemented. Even if a graph library would be used, additional code would have to be written to make the graph structure visible for the graph library. In a subobject-oriented style, these structures are naturally available as defining them requires less effort than writing the corresponding low-level code.

```
class Road {
  Road(City first,City second,Double length) {
    subobject.first(this,first);
    subobject.second(this,second);
    subobject.length(length);
  }
  subobject length Property<Double> {
    export get as getLength;
  }
  subobject first SingleBidi<Road,City> {
    export connect as setFirst, target as getFirst;
    Bidi<City,Road> connectedEnd(City c) = c.roads
  }
  subobject second SingleBidi<Road,City> {
    export connect as setSecond, target as getSecond;
    Bidi<City,Road> connectedEnd(City c) = c.roads
  }
}

class City {
  City() {
    subobject.roads(this);
    subobject.roadNetwork(this);
  }
  subobject roads PassiveSetBidi<City,Road> {
    export targets as getRoads;
  }
  subobject cityToCity WeightedEnd<City,Road> {
    List<Road> intermediates() = outer.getRoads()
    Double weight(Road road) = road.getLength()
    City first(Road road) = road.getSecond()
    City second(Road road) = road.getFirst()
  }
  subobject roadNetwork SimpleWeightedNode<City> {
    export shortestDistanceTo as distanceTo;
    List<WeightedEnd<City,?>> edges() = List(outer.cityToCity)
    WeightedNode<City> node(City city) = city.roadNetwork
  }
}
```

Fig. 41. A subobject-oriented routing application

5 Implementations

We have implemented subobject-oriented programming in two ways. The first implementation is a language extension of Java, called JLo, which is translated to Java code. The second implementation is a library for Python 3 that adds support for subobject-oriented programming by modifying objects and classes at run-time.

JLo [25] is a subobject-oriented extension of Java supported by an Eclipse plugin. The current implementation still uses Java syntax instead of the more concise Scala syntax used in the paper.The JLo compiler generates delegation code and wraps subobject constructor calls in Strategy objects. Special constructors are generated to allow subclasses to refine subobjects. The result is similar to the code in Fig. 5. To preserve multiple inheritance of subobjects, a JLo class is split into a Java interface and Java class. To enable super calls, methods are duplicated and given a unique name. Super calls are first resolved and then rewritten to regular invocations of the generated method that corresponds to the super method.

We also implemented a library [25] for subobject-oriented programming in Python 3. Support for redefining members in an enclosing scope is ongoing work. The library is implemented by two functions: **with_subobjects** and **subobject**, which are used to decorate classes. In Python, the decorated class definition *@expr* ***class*** *C: body* expands to *f=expr;* ***class*** *C: body ; C=f(C).*

The **@subobject**($args$) decorator replaces the nested class with an instance of internal class *Subobject* that records $args$ and the class body. The **@with_subobjects** decorator collects the *Subobject* class members, and generates the delegation code. In addition, it adds a mk_s method for initializing the subobject. Finally, it initializes the ***self****.outer* field of the subobject to point to the outer object.

Fig. 42 shows how a subobject-oriented radio is implemented in Python. To define a subobject s in a class C, a nested class s is defined inside C. Class s is then decorated with **subobject**, and C is decorated with **with_subobjects**. To export members, a name mapping is passed as a set-valued argument with name *exports*. Finally, an object initializes its subobjects by calling the corresponding *mk_X* methods.

```
@with_subobjects
class Radio:
  @subobject(exports={'getValue':'getVolume',
                      'setValue':'setVolume'})
   class volume(BoundedValue): pass

  @subobject(exports={'getValue':'getFrequency',
                      'setValue':'setFrequency'})
   class frequency(BoundedValue): pass

  def __init__(self,vol,freq):
    self.mk_volume(0,vol,11)
    self.mk_frequency(87.5,freq,108)
```

Fig. 42. A subobject-oriented radio in Python

6 Semantics of Subobjects

This section presents the core semantics of subobject-oriented programming by describing how dispatch works for a class-and-subobjects conglomerate in the presence of method and subobject aliasing, method overriding and further binding of subobjects. The semantics addresses two core issues: which method body a path resolves to in a given context; and which context the method body runs in. Given this information, a complete formal semantics for a language based on subobject-oriented programming can be designed in a now-standard fashion.

This semantics is useful, for instance, to help compiler writers correctly implement subobject-oriented programming dispatch mechanism. The semantics also forms the basis of the notion of ambiguity, which any implementation would need to check. If a path resolves to two different method bodies, then a class is ambiguous and an explicit declaration is required to resolve the conflict. Finally, the semantics could form the basis of a type-theoretic foundation of subobject-oriented programming, but this is left for future work.

The following conventions will be used in the semantics: m denotes a method name; t is a class/subobject name. Paths, P, are defined by the following grammar:

$$\begin{aligned} P &::= t \mid \mathsf{this} \mid \mathsf{super} \mid \mathsf{outer} \mid P.P \\ M &::= P.m \\ A &::= \epsilon \mid A.t \end{aligned}$$

P is a path ending in a class or subobject name. M is a path ending in a method name. Paths may also include this, super and outer, where this refers to the current dynamic class/subobject, super refers to the superclass, and outer refers to the surrounding class/-subobject. A path is *pure* if it contains no occurrence of super, this, or outer. A, B, C denote *absolute paths*, consisting only of class/subobject names. Absolute paths refer to locations in code and are therefore used to uniquely identify classes and subobjects.

The semantics is based on several judgements (Fig. 43) that capture the essence of method, subobject and aliasing declarations. These play the role of axioms in the formal system and specific instances can easily be derived from the code. Relations of the form $\Im \in_d A$ capture that some declaration $\Im$ is made in class A. In contrast, relations $\Im \in^* A$ will be introduced later to capture all the declared and inherited (but not overridden) facts about class/subobject A. Aliasing clauses encode the implicit relationship introduced via an **export** clause or through named parameters. For example, a clause **export** a **as** b appearing in subobject P within the context of class/subobject A is modelled by axiom b aliases $P.a \in_d A$.

Both m aliases $M \in_d A$ and t aliases $P \in_d A$ have restrictions. Paths M can be of the form $t.m$, for exporting a method of a subobject into the current interface. This will ensure that the method referred to is one in a direct subobject. The path P is of the form $t'.t''$ to ensure that the aliased subobject is a directly nested subobject of the class/subobject where the declaration occurs. Cases $t \in_d \epsilon$ and t subclasses $t' \in_d \epsilon$ indicate that the declaration is made at the top level, thus, in this case, t and t' are classes and t subclasses t'. For an overriding clause, m overrides $M \in_d A$, M can only be a pure path, and m must be a declared or inherited method or an alias to a method, otherwise it is an error.

$m \mapsto b \in_d A$	method m with body b
m aliases $M \in_d A$	aliasing of m and M
$t \in_d A$	class/subobject t
t aliases $P \in_d A$	aliasing of t and P
t subclasses $t' \in_d A$	subclassing or subobject typing
m overrides $M \in_d A$	overriding of method M

Fig. 43. Judgements: Axiom schemes encoding explicit declarations ($- \in_d A$) in class/subobject A

The following predicates, which can be trivially computed based on the axioms above, will be useful.

$$
\begin{aligned}
m \text{ not declared in } A &\mathrel{\widehat{=}} \neg(\exists b \cdot m \mapsto b \in_d A) \\
t \text{ not declared in } A &\mathrel{\widehat{=}} \neg(t \in_d A) \\
m \text{ not aliased in } A &\mathrel{\widehat{=}} \neg(\exists M \cdot m \text{ aliases } M \in_d A) \\
t \text{ not aliased in } A &\mathrel{\widehat{=}} \neg(\exists P \cdot t \text{ aliases } P \in_d A) \\
m \text{ no new binding } A &\mathrel{\widehat{=}} m \text{ not declared in } A \ \wedge \ m \text{ not aliased in } A
\end{aligned}
$$

The remainder of the semantics will be presented in three interdependent fragments, expressing the inheritance relationships between classes and subobjects (Section 6.1), expressing what is inherited (Section 6.2), and expressing dispatch by resolving path expressions (Section 6.3).

6.1 Inheritance

Next we define a set of rules capturing the inheritance relationship between various classes and subobjects. There are two paths to inheritance: directly via subclassing and indirectly when (potentially) further binding an inherited subobject. The subclassing and inheritance relationships is kept separate, as subclassing is needed for resolving super. The inheritance relation is intransitive and is defined by the two judgements:

A subclasses B	A subclasses B
A inherits B	A inherits from B

These judgements are defined globally using absolute paths, rather than being defined within the context of a specific class (Fig. 44). The first rule for inherits converts subclassing to inheritance. The second rule captures inheritance of subobjects.

6.2 Class/Subobject Contents

The judgements in Fig. 45 describe the contents of a class or subobject, including what is inherited and derived. Method bodies and subobjects also record the location of the corresponding declaration as an absolute path. The judgements for $\downarrow$overrides and overrides$\downarrow$ are used for resolving the left-hand side and the right-hand side of an

$$\frac{t \text{ subclasses } t' \in_d A}{A.t \text{ subclasses } t'} \qquad \frac{P \text{ subclasses } P'}{P \text{ inherits } P'} \qquad \frac{P \text{ inherits } P' \quad (t, \_) \in^* P'}{P.t \text{ inherits } P'.t}$$

Fig. 44. Rules: Subclassing and inheritance

overriding clause. The first is used to determine the location of the overriding method and the second is used to find the end of the alias chain that will dispatch to it. The judgement for dispatches to gives the methods actually available after overriding and inheritance, plus an adjustment to move the 'this' pointer to the correct location within class-and-subobjects conglomerate. FIXEXPLAIN—REVIEWER COMMENT. WHAT IS ROLE of P in dispatcehs to.

$(t, B) \in^* A$	subobject t from source B
$m \mapsto (b, B) \in^* A$	method m with body b from B
m aliases $M \in^* A$	aliasing of m and M
t aliases $P \in^* A$	aliasing of t and P
$M \downarrow$overrides $M' \in^* A$	resolution of M in overriding
(M, b, B) overrides$\downarrow M' \in^* A$	relocating method to M'
m dispatches to $(b, B, P) \in^* A$	dispatch candidate for m. P is used to adjust the dynamic pointer

Fig. 45. Judgements: Declared and inherited class/subobject contents for class A. FIX: These descriptions are terrible.

The judgements in Fig. 46 describe the rules for declared and inherited classes/subobjects and subobject aliasing. In the last rule, t not declared in B prevents subobject t being declared in B. Permitting it would allow non-local further binding, resulting in ambiguity as subobjects could be further bound in more than one code location.

$$\frac{t \in_d A}{(t, A.t) \in^* A} \qquad \frac{(t, P) \in^* A \quad B \text{ inherits } A \quad t \text{ not declared in } B}{(t, P) \in^* B}$$

$$\frac{t \text{ aliases } P \in_d A}{t \text{ aliases } P \in^* A} \qquad \frac{t \text{ aliases } P \in^* A \quad B \text{ inherits } A \quad t \text{ not declared in } B}{t \text{ aliases } P \in^* B}$$

Fig. 46. Rules: Classes/subobjects and subobject aliasing

The rules in Fig. 47 describe method aliasing. An aliasing declaration is inherited even when a new method is declared, in which case the new method also overrides the aliased method. Recall that when two method paths are aliased, they can never be broken apart.

$$\frac{m \text{ aliases } M \in_d A}{m \text{ aliases } M \in^* A} \qquad \frac{m \text{ aliases } M \in^* A \quad B \text{ inherits } A}{m \text{ aliases } M \in^* B}$$

Fig. 47. Rules: Method aliasing

Based on the previous rules, define the following:

$$\text{not aliased } t \in^* A \mathrel{\widehat{=}} \neg(\exists P \cdot t \text{ aliases } P \in^* A)$$
$$\text{not aliased } m \in^* A \mathrel{\widehat{=}} \neg(\exists M \cdot m \text{ aliases } M \in^* A)$$

The most complicated set of rules deal with the interaction between overriding declarations and aliasing. The first collection of rules (Fig. 48) initiates the resolution process based on the declared and inherited overriding declarations. The second collection of rules (Fig. 49) deal with finding an appropriate method body by resolving the LHS of an overrides clause. The third collection of rules (Fig. 50) 'move' this method to the place being overridden, resolving any aliasing along the way. This sets things up so that when performing path resolution to dispatch a method call, one simply follows an alias chain until the end.

$$\frac{m \text{ overrides } M \in_d A}{m{\downarrow}\text{overrides } M \in^* A} \qquad \frac{\begin{array}{c} m \text{ overrides } M \in^* A \quad B \text{ inherits } A \\ m \text{ no new binding } B \end{array}}{m{\downarrow}\text{overrides } M \in^* B}$$

Fig. 48. Rules: Initiate overriding resolution

The first rule in Fig. 49 resolves aliasing of m on the left hand side, if no method is found at M. The second rule deals with a method path starting with a subobject name that is not aliased: the search moves into the subobject. The third rule deals with the case that the subobject name is aliased. The fourth and fifth rules switch to resolving the right-hand side when a candidate method body is found.

The first rule in Fig. 50 removes any outer added in the previous phase and the second rule removes any t from the right-hand side, both adjusting the P component; the adjustments will be used to move the dynamic pointer from the end of the alias chain back to where the method is declared. The third rule expands a subobject alias. The fourth rule expands a method alias.

The rules in Fig. 51 deal with the ultimate dispatch candidates for simple paths consisting of a single name m in the context of some class/subobject. The three cases are when a method is declared in the class/subobject, when a method is inherited but not overridden in any way, and when some external (to the class/subobject) overriding declaration is present.

The diagram in Fig. 52 illustrates the results of applying the rules for ↓overrides and overrides↓. Assume that the facts derived from code are m aliases $t_1.t_2.t_3.n \in^* A$, and m overrides $s_1.s_2.s_3.p \in_d A$, where the first fact would be derived from m aliases $t_1.n \in_d A$, and n aliases $t_2.n \in_d A.t_1$, n aliases $t_3.n \in_d A.t_1.t_2$.

$$\frac{m{\downarrow}\mathsf{overrides}\ M \in^* A \quad m\ \mathsf{aliases}\ M' \in^* A \quad m\ \mathsf{not\ declared\ in}\ A}{M'{\downarrow}\mathsf{overrides}\ M \in^* A} \qquad \frac{t.M'{\downarrow}\mathsf{overrides}\ M \in^* A \quad \mathsf{not\ aliased}\ t \in^* A}{M'{\downarrow}\mathsf{overrides\ outer}.M \in^* A.t}$$

$$\frac{t.M'{\downarrow}\mathsf{overrides}\ M \in^* A \quad t\ \mathsf{aliases}\ P \in^* A}{P.M'{\downarrow}\mathsf{overrides}\ M \in^* A} \qquad \frac{m{\downarrow}\mathsf{overrides}\ M \in^* A \quad m \mapsto (b, B) \in^* A}{(b, B, \epsilon)\ \mathsf{overrides}{\downarrow} M \in^* A}$$

$$\frac{m\ \mathsf{aliases}\ M \in_d A \quad m \mapsto (b, B) \in^* A}{(b, B, \epsilon)\ \mathsf{overrides}{\downarrow} M \in^* A}$$

Fig. 49. Rules: Non-local overriding—finding method body

$$\frac{(b, B, P)\ \mathsf{overrides}{\downarrow}\mathsf{outer}.M' \in^* A.t}{(b, B, t.P)\ \mathsf{overrides}{\downarrow} M' \in^* A}$$

$$\frac{(b, B, P)\ \mathsf{overrides}{\downarrow} t.M \in^* A \quad \mathsf{not\ aliased}\ t \in^* A}{(b, B, \mathsf{outer}.P)\ \mathsf{overrides}{\downarrow} M \in^* A.t}$$

$$\frac{(b, B, P)\ \mathsf{overrides}{\downarrow} t.M \in^* A \quad t\ \mathsf{aliases}\ P' \in^* A}{(b, B, P)\ \mathsf{overrides}{\downarrow} P'.M \in^* A}$$

$$\frac{(b, B, P)\ \mathsf{overrides}{\downarrow} m \in^* A \quad m\ \mathsf{aliases}\ M \in^* A}{(b, B, P)\ \mathsf{overrides}{\downarrow} M \in^* A}$$

Fig. 50. Rules: Non-local overriding—moving to end of alias chain

$$\frac{m \mapsto b \in_d A}{m\ \mathsf{dispatches\ to}\ (b, A, \epsilon) \in^* A}$$

$$\frac{m \mapsto (b, A) \in^* A \quad B\ \mathsf{inherits}\ A \quad m\ \mathsf{no\ new\ binding}\ B}{m\ \mathsf{dispatches\ to}\ (b, A, \epsilon) \in^* B}$$

$$\frac{(b, B, P)\ \mathsf{overrides}{\downarrow} m \in^* A \quad \mathsf{not\ aliased}\ m \in^* A}{m\ \mathsf{dispatches\ to}\ (b, B, P) \in^* A}$$

Fig. 51. Rules: Dispatch candidates

The first intermediate result is

$$n{\downarrow}\mathsf{overrides\ outer.outer.outer}.s_1.s_2.s_3.p \in^* A.t_1.t_2.t_3,$$

giving the result after resolving the aliasing, thus n will be the name of some actual method with body b_n declared in some class/subobject B. From this it immediately follows that:

$$(b_n, B, \epsilon)\ \mathsf{overrides}{\downarrow}\mathsf{outer.outer.outer}.s_1.s_2.s_3.p \in A.t_1.t_2.t_3.$$

The second intermediate result is

$$(b_n, B, \mathsf{outer.outer.outer}.t_1.t_2.t_3)\ \mathsf{overrides}{\downarrow}p \in^* A.s_1.s_2.s_3$$

which gives relates the overriding method body (b_n, B) with the place where the overriding occurs, namely, $A.s_1.s_2.s_3.p$. The resulting dispatch candidate will be

$$p\ \mathsf{dispatches\ to}\ (b_n, B, \mathsf{outer.outer.outer}.t_1.t_2.t_3) \in^* A.s_1.s_2.s_3.$$

The additional component, $\mathsf{outer.outer.outer}.t_1.t_2.t_3$, is applied to a dynamic pointer during dispatch to move it to the correct location (following the dotted arrow in Fig. 52).

6.3 Path Resolution

The semantics of path resolution is based on three judgements (Fig. 53). The judgements describe how to lookup a method body in a given context, how to evaluate a method body in a given context, and how to resolve a path to a subobject.

Resolving a path involves navigating around a class and its respective subobjects and their superclasses. Doing so requires keeping track of two 'pointers', one for the dynamic class/subobject of 'this', the other for the static code location where the current class/subobject is found. This will be represented by $\langle D, S\rangle$, where D is the dynamic part and S is the static part of the location.

The rules in Fig. 54 deal with method dispatch. We assume that local method call paths begin with this, as it plays the role in the semantics to ensure that the most specific static class/subobject is considered. All other components of the dispatch is done based on the static pointer. The first rule finds the candidate method associated with the equivalence class of paths to the method. A path of the form $D.\mathsf{outer}^n.P'$, introduced when P is appended to D in the first rule, can be reduced to an absolute path by iterating the following equivalence $D.t.\mathsf{outer}.P = D.P$ until all occurrences of outer have been eliminated. The rules for evaluating a method body have been omitted, but can be added in a straightforward fashion; the key point of interest is that the context in which that body is run is, namely, some pair $\langle D, S\rangle$.

The rules for path resolution are given in Fig. 55. Each part of a path results in an incremental change to both the dynamic and static parts of the context. The rules are ambiguous in the case that a subobject and an alias with the same name coexist together. The compiler must rule out such cases. The rule for this selects most specific textual class for a given dynamic class to start the search. The two rules for subobject name t look up the name in the given static path or replace it with the path it aliases, respectively. The rule for super finds the superclass of the current static code location, and the rule for outer finds the surrounding class/subobject of the current static code location. The final rule describes how to resolve longer paths.

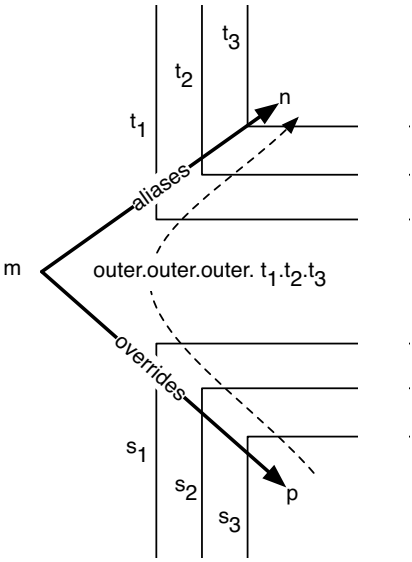


Fig. 52. Example: Overriding resolution, assuming m overrides $s_1.s_2.s_3.p$ and m overrides $s_1.s_2.s_3.p$ are declared in A

$\langle D, S\rangle\ M \Longrightarrow v$	method M evaluates to v
$\langle D, S\rangle\ b \Longrightarrow v$	body b evaluates to v (omitted)
$\langle D, S\rangle \xrightarrow{P} \langle D', S'\rangle$	P moves from $\langle D, S\rangle$ to $\langle D', S'\rangle$

Fig. 53. Judgements: Resolution and evaluation — D, S are absolute paths

$$\frac{m \text{ dispatches to } (b, B, P) \in^* S \quad \langle D.P, B\rangle\ b \Longrightarrow v}{\langle D, S\rangle\ m \Longrightarrow v}$$

$$\frac{\langle D, S\rangle \xrightarrow{P} \langle D', S'\rangle \quad \langle D', S'\rangle\ m \Longrightarrow v}{\langle D, S\rangle\ P.m \Longrightarrow v}$$

Fig. 54. Rules: Method path resolution

These rules capture the essence of the composition mechanism of subobject-oriented programming. Providing a complete semantics for the full language including type safety theorems and their proof is a topic for future work.

$$\frac{D = A.t \qquad (t, P) \in^* A}{\langle D, S\rangle \xrightarrow{\mathsf{this}} \langle D, P\rangle} \qquad \frac{(t, P) \in^* S}{\langle D, S\rangle \xrightarrow{t} \langle D.t, P\rangle}$$

$$\frac{t \text{ aliases } P \in^* S \quad \langle D, S\rangle \xrightarrow{P} \langle D', S'\rangle}{\langle D, S\rangle \xrightarrow{t} \langle D', S'\rangle} \qquad \frac{S \text{ subclasses } S'}{\langle D, S\rangle \xrightarrow{\mathsf{super}} \langle D, S'\rangle}$$

$$\frac{}{\langle D.t, S.t'\rangle \xrightarrow{\mathsf{outer}} \langle D, S\rangle} \qquad \frac{\langle D, S\rangle \xrightarrow{P} \langle D'', S''\rangle \quad \langle D'', S''\rangle \xrightarrow{P'} \langle D', S'\rangle}{\langle D, S\rangle \xrightarrow{P.P'} \langle D', S'\rangle}$$

Fig. 55. Rules: Path Resolution

7 Related Work

For a discussion on aspect-oriented programming, mixins, traits, and non-conformant inheritance, we refer to Section 2.

Subobject-oriented programming is based on the *component relation* that we previously introduced for composition of classes [26]. Subobjects refine the component relation in a number of ways. With the component relation, member redefinitions were written in the composed class, and then wired into the subobject with overrides clauses. This lead to scattering of subobject members throughout the composed class. With subobjects, such redefinitions can be written directly in the subobject. This eliminates the overrides clauses, and significantly improves readability. The dedicated parameter mechanism for connecting components was removed and replaced by using methods to connect subobjects to each other. Switching to Scala syntax for method definitions removed most of the overhead. Most importantly, our previous work lacked support for object initialization and super calls, both of which are essential in real programming languages. In addition, our previous work was not implemented and demonstrated only very basic subobjects.

Reppy and Turon present trait-based metaprogramming [19], which is very similar to non-conformant inheritance in Eiffel. Traits are checked at compile-time and then inlined. The name of each method is a parameter that is used to rename it in the reusing class. Because the renaming is deep, this allows proper resolution of name conflicts, but it also requires a lot of work. If two traits of the same kind are used, all of their methods must either be renamed or excluded. Contrary to Eiffel, sharing is not the default policy. Instead, a type error is reported when multiple methods with the same signature are inlined. A trait can contain fields, which must be initialized by the constructor of a class that uses the trait. The special type *ThisType* is used to impose constraints on the class that reuses a trait. This is similar to requirements in regular traits, and to abstract methods in subobjects. A method of a trait can override a method of the outer class. The original method is available as *outer.m(...)*. Once a trait method overrides a method of the outer class, it is considered to be locally defined. As a result, it can again be overridden by another trait method. The resulting concatenation of traits is similar to mixin-based inheritance. Contrary to subobjects, traits cannot be used as separate objects, prevent certain kinds of reuse.

Object layout in C++ [22] is often described in terms of subobjects, where each inherited class forms a subobject. The key difference with our subobjects is that in our approach subobjects are placed in separate namespaces, which avoids many of the problems of C++. More concisely, C++ implements subobject-based inheritance, whereas subobject-oriented programming is about composition of subobjects. Refinement of subobjects is not possible in C++. Our approach uses the rule-of-dominance of C++ to resolve conflicts if a single best candidate is available.

Languages that implement further binding include Beta [17], gbeta [10], and Tribe [5]. In these languages, further binding applies to nested classes, which can be used to create objects of the same family. In our approach, further-binding applies to nested subobjects, which define a static part of the composed class. Virtual classes [9,11] do resemble subobjects, but their purpose is completely different. Virtual classes support family polymorphism, whereas subobjects support composition of classes. As such, neither technique can be used as a substitute for the other. Any number of objects can be constructed from virtual classes and path-dependent types are required to ensure that certain object belong to the same family. With subobjects on the other hand, there is only one "instance" of each subobject per outer object. Subobject names serve only to avoid conflicts and access the parts. Therefore, path-dependent types and the associated complexities are not needed.

Subject-oriented programming [13] differs from subobject-oriented programming in the purpose of the composition. Composition in subject-oriented programming is about the separation of concerns, and thus more related to aspects-oriented programming and family polymorphism, whereas subobject-oriented programming is about composing classes, and is thus more of a refinement of classical object-oriented programming. Both approaches are complementary, since the different view-points could implement parts of their customization with subobjects.

Madsen and Møller-Pedersen [16] introduced part objects in the context of Beta for better structuring code. A part object is a locally defined object. Part objects know their location, which in our setting is given by the `outer` reference. Their motivation is similar to ours, but their language lacks the constructs for composing and refining the part objects (subobjects) as ours does—more precisely, these need to be coded up in regular Beta code. Beta does however also offer refinement/further binding of nested classes, which is something few other languages support.

A split object [2] consists of a collection of *pieces* which represent particular viewpoints or roles of the split object, have no identity, and are organized in a delegation hierarchy. Invoking methods is done by selecting a viewpoint to send the message to. The main difference with subobjects is that subobjects are used to build classes, whereas pieces are used to model different viewpoints on a class. The substructures in both approaches have an opposite order with respect to overriding. A piece overrides methods from its enclosing pieces and class, whereas enclosing subobjects and the composed class override methods of more deeply nested subobjects. In addition, members in pieces cannot be merged, whereas members from different subobjects can be merged. Finally, pieces are added dynamically, whereas subobjects are declared statically.

Blake and Cook [3] propose to add part hierarchies to object-oriented languages. These resemble nesting of subobjects, but the proposed implementation does not include the advanced features for refining subobjects.

8 Conclusion

Existing object-oriented and aspect-oriented techniques do not offer the features to build a class using other classes as building blocks. Instead of being encapsulated in a class and reused, cross-cutting structural code for general purpose concepts such as associations and graphs must be implemented over and over again.

Subobject-oriented programming improves on object-oriented programming by allowing programmers to easily build a class from other classes. This work improved on our previous work in a number of ways. We defined subobject initialization and super calls. We improved the adaptability of subobjects by using refinement instead of overriding. We improved the readability of subobjects using a more object-oriented syntax and removed the functional style parameter mechanism. In addition, we have implemented subobject-oriented programming as a language extension to Java [12], and as a library in Python 3 [20]. Finally, we have also developed a library of classes that demonstrates the advanced possibilities of subobject-oriented programming.

References

1. Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K.: An overview of caesarJ. In: Rashid, A., Akşit, M. (eds.) Transactions on AOSD I. LNCS, vol. 3880, pp. 135–173. Springer, Heidelberg (2006)
2. Bardou, D., Dony, C.: Split objects: A disciplined use of delegation within objects. In: Proceedings of OOPSLA 1996, pp. 122–137. ACM Press (1996)
3. Blake, E., Cook, S.: On including part hierarchies in object-oriented languages with an implementation in smalltalk. In: Bézivin, J., Hullot, J.-M., Lieberman, H., Cointe, P. (eds.) ECOOP 1987. LNCS, vol. 276, pp. 41–50. Springer, Heidelberg (1987)
4. Bracha, G., Cook, W.: Mixin-based inheritance. In: Proceedings of OOPSLA/ECOOP 1990, pp. 303–311 (1990)
5. Clarke, D., Drossopoulou, S., Noble, J., Wrigstad, T.: Tribe: A simple virtual class calculus. In: Proceedings of AOSD 2007, pp. 121–134 (2007)
6. Colnet, D., Marpons, G., Merizen, F.: Reconciling subtyping and code reuse in object-oriented languages: Using inherit and insert in SmartEiffel, the GNU Eiffel compiler. In: Morisio, M. (ed.) ICSR 2006. LNCS, vol. 4039, pp. 203–216. Springer, Heidelberg (2006)
7. Van Cutsem, T., Bergel, A., Ducasse, S., De Meuter, W.: Adding state and visibility control to traits using lexical nesting. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 220–243. Springer, Heidelberg (2009)
8. Ducasse, S., Wuyts, R., Bergel, A., Nierstrasz, O.: User-Changeable visibility: Resolving unanticipated name clashes in traits. In: OOPSLA, pp. 171–190 (2007)
9. Ernst, E.: Family polymorphism. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 303–326. Springer, Heidelberg (2001)
10. Ernst, E.: Higher-order hierarchies. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 303–329. Springer, Heidelberg (2003)
11. Ernst, E., Ostermann, K., Cook, W.R.: A virtual class calculus. In: POPL, pp. 270–282 (2006)

12. Gosling, J., et al.: The Java Language Specification, 2nd edn. Addison-Wesley Longman Publishing Co. Inc. (2000)
13. Harrison, W.H., Ossher, H.: Subject-oriented programming (a critique of pure objects). In: Proceedings of OOPSLA 1993, pp. 411–428 (1993)
14. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–354. Springer, Heidelberg (2001)
15. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997 — Object-Oriented Programming. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
16. Madsen, O.L., Møller-Pedersen, B.: Part objects and their location. In: Proceedings of TOOLS 1992, pp. 283–297 (1992)
17. Madsen, O.L., Møller-Pedersen, B., Nygaard, K.: Object-Oriented Programming in the Beta Programming Language. Addison-Wesley (1993)
18. Odersky, M., Zenger, M.: Scalable component abstractions. In: Proceedings of OOPSLA 2005, pp. 41–57 (2005)
19. Reppy, J., Turon, A.: Metaprogramming with traits. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 373–398. Springer, Heidelberg (2007)
20. Rossum, G.V., Drake, F.: Python 3 Reference Manual. CreateSpace (2009)
21. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.: Traits: Composable units of behavior. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 248–274. Springer, Heidelberg (2003)
22. Stroustrup, B.: The C++ programming language, 2nd edn. Addison-Wesley Longman Publishing Co. Inc., Boston (1991)
23. Technical Group 4 of Technical Committee 39. ECMA-367 Standard: Eiffel Analysis, Design and Programming Language. ECMA International (2005)
24. van Dooren, M., Clarke, D.: Subobject transactional memory. In: Sirjani, M. (ed.) COORDINATION 2012. LNCS, vol. 7274, pp. 44–58. Springer, Heidelberg (2012)
25. van Dooren, M., Jacobs, B.: Implementations of subobject-oriented programming (2011), `http://people.cs.kuleuven.be/marko.vandooren/subobjects.html`
26. van Dooren, M., Steegmans, E.: A higher abstraction level using first-class inheritance relations. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 425–449. Springer, Heidelberg (2007)

Verification of Open Concurrent Object Systems*

Ilham W. Kurnia and Arnd Poetzsch-Heffter

University of Kaiserslautern, Germany
{ilham,poetzsch}@cs.uni-kl.de

Abstract. A concurrent object system is a dynamically growing collection of concurrently interacting objects. Such a system is called *open* if the environment of the system is unknown. Proving properties about open systems is challenging because the properties must be shown to hold for all possible environments of the system. *Hierarchical* reasoning, which infers properties of large components from the properties of smaller subcomponents, is a key enabler to manage the reasoning effort.

This chapter presents an approach to hierarchically specify and verify open concurrent object systems. We introduce a core calculus for concurrent object systems. The behavior of such a system is given by a standard operational semantics. To abstract from the internal representation of the objects, we develop an alternative trace-based semantics that captures the behavior in terms of the communication traces between the objects of the system and the environment. The main advantage of the trace-based model is its extendability to components and open systems while remaining faithful to the operational model. The specification approach is also directly based on the traces and supports hierarchical reasoning using the following two central features. *Looseness* allows specification refinement. *Restriction* allows expressing assumptions on the environment. Finally, we provide a Hoare-style proof system that handles the given specifications.

1 Introduction

Modern software systems are becoming more and more concurrent and distributed. The main reasons are that they have to exploit the resources of many-core computer architectures and realize distributed applications. Unfortunately, concurrency increases the complexity of software and is often a source of errors. Whereas testing is a successful technique to improve the quality of sequential software, testing is much less appropriate for concurrent software because of its high degree of nondeterminism. Here, verification techniques come to help (see, e.g., [8, Chap. 1], [49, Chap. 1]).

Object-oriented framework is a recommended choice for building concurrent and distributed systems [31]. There are many approaches how concurrency can be introduced into the object-oriented framework (see, e.g., a survey by [47]). In this chapter, we describe a technique to verify functional properties of *concurrent object* systems [63]. A concurrent object (the intrinsic concurrency model of the modeling language ABS [33]

* This work is partially supported by the EU project *FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models*.

E. Giachino et al. (Eds.): FMCO 2012, LNCS 7866, pp. 83–118, 2013.

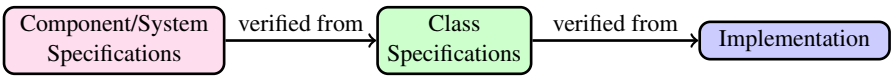


Fig. 1. Two-tier verification

discussed in this volume) is a computational entity whose state is fully encapsulated. It communicates with other objects exclusively by exchanging asynchronous messages. Within each object, only a single, cooperatively scheduled thread is running at any time. Thus, the behavior of single objects[1] can be verified using sequential techniques. Similar to popular programming languages such as Java and C++ and also ABS, classes are used as to describe the behavior of objects. The combination of encapsulation, concurrency among objects, and sequential execution within objects grants scalability as explained, e.g., in [6,28]. Concurrent objects are similar to actors [3]; the two approaches mainly differ in how behavior changes and interfaces are expressed.

There are already a number of verification techniques for concurrent objects available (see, e.g., [23,11,5,20,22]). The technique presented in this chapter centers around two specific aspects: *openness* and *hierarchical reasoning*. Openness allows the verification of components without knowing the environments in which they are used, while hierarchical reasoning allows verifying larger components from smaller ones in a hierarchical way. The technique follows a two-tier verification approach proposed by, for example, Misra and Chandy [43] and Widom et al. [62], as shown in Fig. 1. In the first tier, the behavior of a class is specified and verified against its implementation. That is, the program code of the class is used to prove that objects of the class have a specified behavior at their interfaces. This issue is *not* treated in this chapter. It is handled, e.g., in [5,20,22]. Here we focus on proving the specification of larger components from the specifications of smaller ones where the smallest components are classes. The verification technique is *modular*, i.e., the specification of smaller components is sufficient to prove larger ones.

Essential to our verification approach is the notion of "component". Components according to this notion should satisfy at least the following requirements:

- The specification technique for components must be applicable to classes, because they are the base of the hierarchical approach.
- Components should allow the construction of increasingly large systems.
- Components need clear semantical interfaces that can hide internal behavior.

In our approach to components, we follow a basic idea from software component models such as OSGi [60] and COM [40]. A component has an activator class AC. A component instance is created by creating an object X_{AC} of class AC. At run time, the component instance consists of the set $O(X_{AC})$ of objects transitively created by X_{AC}. A communication of $O(X_{AC})$ with the environment happens when an object in $O(X_{AC})$ sends a message to an object outside of $O(X_{AC})$ or vice versa. Of course, the exact formation of a component instance is influenced by how the environment interacts with the component. A component is then by nature an open system.

[1] Henceforth we refer to concurrent objects simply as objects.

The specification of a component abstracts from the *internal* messages and objects. It only describes the communication of the component with the environment. Composition is done by developing new components using other components for their implementation. It is similar to procedural programming in that a procedure calls other procedures and in that the specification abstracts from procedure-local state and local calls.

To specify the desired properties of components we need to define their semantics. We first introduce an operational semantics of the classes, that is, of the base components, by means of transition systems and core operational rules. From the operational semantics, we derive a trace semantics. That is, we represent an execution of an object system by a trace of observable events [30]. The advantage of dealing only with traces is the abstraction from the actual state representation of the system. The semantics of objects and components is expressed by *trace sets*.

Based on the trace semantics, we develop a specification technique relating input traces to output traces. Input (output) traces represent events an object or a component receives (produces). Distinguishing input and output traces is done based on the trace set. Formally, a specification consists of a finite set of Hoare-like triples [29]. A triple $\{p\}\ D\ \{q\}$ means that if an input trace satisfies p, component D produces output traces satisfying q. The component D represents either the behavior of single objects of class C or the (external) behavior of groups of objects with an initial object of class C. A triple specifies the behavior of a component only for inputs satisfying p. This input condition provides assumptions about how the component is used and help to focus the reasoning.

This chapter follows closely on the authors' previous work [37], which explains the specification and verification approach in detail. We focus here on the connection between an operational semantics of concurrent object systems and its trace semantics in the open system context.

Chapter Structure. In Sect. 2 we explain the setting and provide a running example used in this chapter. Section 3 deals with the core operational semantics of concurrent object systems. Section 4 describes how components and their trace semantics can be extracted from the operational semantics. Section 5 presents the specification and verification technique with the help of the running example. We conclude this chapter with some discussion how this approach can be extended. Related work and discussion on the subject of each section are provided at the end of each section.

2 Setting and Running Example

To have a sufficiently clear background for the following discussion on verification, we informally introduce a core concurrent object language ActJ together with an example for illustrating our approach. ActJ can be thought of as a stripped down version of ABS [33][2]. ABS is an object-oriented modeling language that permits, among others, actor-style synchronous communication. ActJ focuses on the actor concurrency layer of ABS. It features a Java-like syntax, interfaces, class-based programming, asynchronous

[2] Information about ABS can also be found in the chapter by Reiner Hähnle in this volume [27]. Various design choices of ActJ can be inferred from the design choices of ABS.

method calls without returns, concurrent objects, abstract data types with first-order side-effect free functions and typed references. This language is used to describe a variant of an industrial distributed database system [7].

2.1 ActJ

Objects in ActJ is described by means of classes. A class is a template how an object is represented and behaves according to the representation. Through fields a class defines what data can be in an object and through methods a class defines how operations can be applied on the object. An object can be created by instantiating the class. A method implementation may be equipped a *guard*, regulating when a method may be executed (explained later). All fields are private to individual instances, providing strict encapsulation on instance level. Object manipulation described exclusively through methods, which are public. Methods contain a list of statements. A statement can be an object creation, an asynchronous method call, a conditional statement, field assignments and a sequential composition of statements. Expressions contained in a statement are pure functions and abstract data types such as lists can be used.

ActJ follows the principle of *programming to interfaces* [25], where the declared types of objects are interface types. Interfaces consists of a number of method signatures, stating the name and parameters of operations that can be applied on the object. To focus on the core behavior of the objects, subtyping is not introduced in ActJ. For the same reason, each class implements exactly one interface. As a simplification, method signatures do not contain have a return type. Hence, computation result that needs to be communicated back is delivered through making method calls. Examples of ActJ programs are given in Figs. 2 and 3, which is explained in the following subsection.

2.2 Running Example

As a running example, we use a variant of the client-server setting treated by Arts and Dam [7]. The server receives requests from the client, where each request contains a query. The server system responds to the requests with the appropriate computation results. To serve each request, the server creates a worker and passes on the query to be computed. A query can be divided into multiple chunks, therefore concurrent processing of a request can be introduced as follows. Before each worker processes the first chunk of the query, it creates another worker to which the rest of the query is passed on. When the computation of the first query chunk is finished, the worker merges the previous result with this computation result and propagates the merged result to the next worker. Eventually all chunks of the query are processed, and the last worker sends back the final result to the client. The client identity must be passed around, so that the last worker can return the query computation result to the client.

An implementation of the server side is given in Fig. 2. It consists of two classes: `Server` and `Worker`. The `Server` class implements the `IServer` interface, whereas `Worker` implements `IWorker`. These classes use the `IClient` interface so they can communicate with the client. A client can send a request to the server by calling the `serve` method. A query is represented by the `Query` data type. The server processes the request by creating a new `Worker` object, passing on the query to this new worker

```
interface IClient {
  response(Value);
  link(IServer);
}
interface IServer {
  serve(IClient c, Query y);
}
interface IWorker {
  do(Query y);
  propagate(Value v, IClient c);
}

class Server implements IServer {
  serve(IClient c, Query y) {
    // querySize(y) ≥ 1
    IWorker w = new Worker;
    w.do(y);
    w.propagate(null, c);
  }
}
```

```
class Worker implements IWorker {
  Value myResult = null;
  IWorker nextWorker = null;
  do(Query y) {
    if (querySize(y) > 1) {
      nextWorker = new Worker;
      nextWorker.do(restQuery(y));
    }
    myResult = compute(fstQuery(y));
  }
  propagate(Value v, IClient c)
    guard myResult != null {
    if (nextWorker == null) {
      c.response(merge(myResult, v));
    } else {
      nextWorker.propagate(
              merge(myResult, v), c);
    }
  }
}
```

Fig. 2. Server-worker implementation in ActJ

and then initiate the result propagation. The execution of the `serve` method acts as an example how objects can interact with other objects in ActJ.

There are two ways objects can interact with each other: creating new objects and calling methods of other objects. Object creation is represented in ActJ by the execution of a statement of the form `new C`, where `C` is a class name. Object creation is *blocking*, meaning that the execution cannot continue to the next statement before the object is created. A method call is produced when a statement of the form `r.m(`$\overline{\mathtt{p}}$`)` is executed. This statement sends the message `m(`$\overline{\mathtt{p}}$`)` to the receiver object `r` where `m` is the method name with a list of parameters $\overline{\mathtt{p}}$. The parameters can be data values or object identities. A method call is *non-blocking*; execution directly continues with the next statement. Thus, in general, a method call leads to concurrent behavior. The asynchronous nature of method calls leads to the possibility that when two method calls are sent one after the other, they may arrive at their respective destination in a different order. For each of the calls an object receives, it has a *body* that describes how it reacts to a method call.

The `Worker` class illustrates how **`guard`** is used. The class has two methods `do` and `propagate` and two fields `myResult` and `nextWorker`, both initialized to **`null`**. The `do` method checks (via the function `querySize`) whether the query the worker has to do can be split to further subqueries. If the query is splittable, a new worker is created and the reference of this new worker is stored in the `nextWorker` field. The new worker is then sent the rest of the query (via the function `restQuery`) the current worker is not handling. The current worker then proceeds on computing the result of the first subquery (or the entire query if it is not splittable). The computation is done using the function `compute` and when it is completed, the result is stored in `myResult`.

```
class Client implements IClient {
  IServer s = null;
  link(IServer server) {
    s = server;
    s.serve(this, randomQuery());
  }
  response(Value v) {}
}
```

Fig. 3. A sample `Client` in ActJ

The `propagate` method states how a worker propagates the result of its computation. The **`guard`** ensures that the result propagation is carried out *only* when the worker's part of the query computation is finished. If the worker is the last worker handling the query (indicated by the non-existence of the next worker object), a `response` call is sent to the client, whose reference is passed on with the call. This response contains the merged result of the worker's computation of the subquery it handled with the value passed on with the call. Otherwise, the merged result is passed on to the next worker. In short, the series of `propagate` calls, initiated by the server, collect and merge the results of computing the different query chunks and send the clients the response.

It remains to explain what precisely happens when a method call is received. We assume that objects, similar to actors [3], have an unbounded input queue and are input-enabled (cf. [39, p. 257]); i.e., objects can always accept new input. An object is always in one of two modes: *idle* and *active*. Whenever an object is idle, meaning it has just been created or finished executing a method, it selects a method call from its input queue to be executed. If the queue is empty, then the object is idle. In other cases, an object is active. When a method call is selected, it is removed from the queue. During this selection process, the guards of the corresponding method definitions are evaluated. Method definitions that do not have a guard are equivalent to method definitions whose guard is always **`true`**. Method calls are then *selected* from the queue primarily in a First-In-First-Out manner. In other words, the first pending method call in the queue whose guard is evaluated to **`true`** is selected. This selection process guarantees (weak) fairness, meaning that a method call whose guard is infinitely often evaluated to **`true`** will eventually be picked for processing. The presence of guards gives an object control over the execution of incoming method calls.

We assume appropriate definitions for the pure functions used in the `Worker` class. In addition, the following properties are assumed.

```
querySize(t) ≥ 1
querySize(t) > 1 ⟹ compute(t) = merge(compute(fstQuery(t)),
                                      compute(restQuery(t)))
querySize(t) = 1 ⟹ compute(t) = compute(fstQuery(t))
merge(null, v) = v
```

A query consists of at least one chunk; computing a non-primitive query is the same as merging the result of computing the first query with the computation of the rest of the query; computing a single query chunk is the same as computing the first query of the chunk; merging with **`null`** with some value `v` results in the value `v`.

One possible way to use the server is by constructing a client whose implementation can be seen in Fig. 3. Through the `IServer` interface, the client sends some random query (by means of some built-in method `randomQuery`) to a server when the client is linked to the server. When it receives a response, it does nothing. When a client is

created and the client is linked to a server, the client makes a request to the server. Then the server creates a number of worker to handle the query, and the last worker returns the query computation result back to the client.

The running example highlights the features of concurrent objects, namely asynchronous method calls, sequential execution of each method and encapsulation of internal state. Moreover, the example shows unbounded object creation, a non-trivial aspect to handle in verifying functional behavior. The unbounded object creation is caused by the lack of knowledge on the server side with respect to the number of subqueries a query can be split into. Recall that a subquery is handled by a separate worker.

2.3 Discussion

In object-oriented programming, it is common for objects to interact with other objects by field access. In the concurrent setting, however, allowing field access also means that explicit signaling between objects is needed to indicate that an object is being modified. This leads to complex techniques, such as [1,14], to understand what exactly is going on with the objects. Fields in ActJ are *object private*, meaning each object can only see and modify its own fields; even fields of other objects of the same class are inaccessible. Putting the plug on field access allows an object to rely on the stability of the values of its fields during a method execution. More discussion on the mechanisms of concurrent interaction between objects is available in [32].

In general, it is desirable to be able to synchronize the progress between different method call executions (cf. `await` construct in ABS). An object is then allowed to *multitask*, meaning that the execution of several messages can be in progress (not just residing in the queue). This concept does not appear in our running example (only partially through the use of `guard`). To accommodate this concept, the object idle mode needs to be extended to the state when a method call execution awaits for certain synchronization. Also, objects need to be equipped with a more specific scheduler.

Methods in ActJ do not have return values except the implicit indication that an execution of a method is finished. Returning specific values is emulated by sending a method call. If an object needs any information from other objects or wants to manipulate other objects, it must be done by calling methods of the other objects. A more elegant way to handle returns is to use futures [9] as in ABS.

3 Core Operational Semantics

To characterize the behavior of a concurrent object system, we first look into the behavior of objects and how they interact with each other. Following the design of modeling languages such as Creol [34] and ABS [33], classes provide blueprints how their instances, i.e., the objects, behave. Instead of directly providing the operational semantics based on the code structure, the classes are modeled using parameterized transition systems. This approach allows for a simple run-time configuration model. We use a slightly less faithful model in terms of the communication between objects to capture the behavior of classes. This model is expressive enough for showing the connection between operational semantics of concurrent object systems and their trace semantics. The trace semantics is used as the basis for specification and verification.

3.1 Notation

We first define the notation used in this chapter. Capital letters represent sets. Typical elements of sets and variables are represented by small letters. Constants are written using a typewriter font, usually to indicate the connection with the running example. Exceptions to the notation are clearly noted. A summary of numerous helper predicates and functions used in the semantics description is given in Table 1 as a quick reference.

We use the data structure $Seq\langle\mathbf{T}\rangle$ to represent finite sequences, with **T** denoting the type of the sequence elements. An empty sequence is denoted by $[]$ and $\cdot$ represents sequence concatenation. The function $Pref(s)$ yields the set of all prefixes of a sequence s. The projection operator $s{\downarrow_T}$ produces the longest subsequence[3] of the sequence s that contains sequence elements in T of type **T**. The projection operator can be refined by considering the structure of **T**.

3.2 Classes

We start the description of our formal model by defining the basic sets. Let **O** be the set of all object identities, **CL** the set of all classes, **M** the set of messages that can be communicated between objects and **D**, disjoint from **O**, the set of data values. We use object identities to represent both the object and its actual identity. The function $class(o)$ gives the class of an object o. A message m can either be an object creation `new` C or a method

Table 1. Helper predicates and functions

Predicate/Function	Description
$Pref(s)$	The set of all prefixes of sequence s.
$class(o)$	Returns the class of object o.
$isMtd(m)$	Checks if message m is a method call.
$callee(e)$	Returns the callee of event e.
$caller(e)$	Returns the caller of event e.
$msg(e)$	Returns the message of event e.
$acq(t)$	Returns the accumulated object identities exposed in each method call event in trace t.
$cr(t)$	Returns the set of objects created in trace t.
$extMtd(C)$	Returns the set of method calls that can be received by objects of class C.
$t{\downarrow_O}$	Projects trace t to non-external events of a set of objects O. The operator can also take *callee* or *caller* as an extra parameter.
$exposed(t, O)$	$acq(t{\downarrow_{O,callee}}) \cup cr(t{\downarrow_{O,caller}})$
$idx(t, C)$	Extracts the identities of objects transitively created by the initial object of class C from trace t.
$sup(t)$	Returns the event core sequence of t.
$split(t, L)$	Splits t into input and output traces (ti, to) based on local objects L.
$bound(T)$	Extracts the largest visible subset of local objects from a component trace set T.

[3] A sequence s is a *subsequence* of another sequence s' if s can be derived from s' by deleting some elements of s' while preserving the order of the remaining elements [26, p. 4].

call $mtd(\overline{p})$, where C is a class, mtd denotes some method name and $\overline{p}$ is a list of parameters. A parameter may be a data value $d \in \mathbf{D}$ or an object identity. The predicate $isMtd(m)$ checks if the message m is a method call. The messages are the observable units of communication exchange between objects.

The set of *events* $\mathbf{E}$ is built on the messages. An event $e \in \mathbf{E}$ represents the occurrence of a message $m = msg(e)$ being sent by the *caller* object $o_1 = caller(e)$ to the *callee* object $o_2 = callee(e)$. If m is a creation message, o_2 will be the name of the newly created object while o_1 is its creator. Textually an event e is represented as $o_1 \rightarrow o_2.\texttt{new}\ C$ or $o_1 \rightarrow o_2.mtd(\overline{p})$ when the message is an object creation or a method call, respectively. With respect to some group of objects $O \subseteq \mathbf{O}$, an event $o_1 \rightarrow o_2.m$ can be classified into *internal event*, if $o_1, o_2 \in O$; *external event*, if $o_1, o_2 \notin O$; *input event*, if $o_1 \notin O$ and $o_2 \in O$; and *output event*, if $o_1 \in O$ and $o_2 \notin O$. To collect the (finite number of) object identities occurring in the parameter list of a method call, we define a function $acq(mtd(\overline{p}))$, short for *acquaintance*.

To describe the behavior of a specific object, only the callee and message information of an event are needed. Following the design choice of ABS, the caller information is transparent from the callee of a method call. In the same way, an object does not know who creates it, unless this information is passed on explicitly. Therefore, if we focus only on a specific object, the caller information becomes redundant. Eliminating the caller from an event produces an *event core*. The set of event cores $\mathbf{EC}$ are derived from the set of events $\mathbf{E}$ by removing the caller from events in $\mathbf{E}$. The functions *msg* and *callee* applied to event cores are defined as for events.

The class of an object characterizes how the object behaves. One way to model classes is by transition relations. We model classes using two transition relations: one describing what an object does when it receives a message and the other describes what an object does when it reaches a certain state. As the model should reflect common invariants on classes, it also contains a function that tells which objects are known to the class instance. By restricting the model via this function, the class model guarantees, for example, that method calls can only be sent to known objects.

Definition 1 (Class). *A* class C *is a parameterized tuple* $\langle Q, q^0, \rho, \alpha, \kappa \rangle(\texttt{this})$ *where*

- Q *is the set of states,*
- $q^0 : O \rightarrow Q$ *is the parameterized initial state,*
- $\rho : \mathbf{M} \times Q \times Q$ *is the message receive relation,*
- $\alpha : Q \times \mathbf{EC} \times Q$ *is the action relation, and*
- $\kappa : Q \rightarrow 2^{\mathbf{O}}$ *is the function mapping a state to a set of known objects,*

where for each state $q, q' \in Q$*, message* $m \in M$ *and event core* $\mathsf{e} \in \mathbf{EC}$*, representing* $\langle m, q, q' \rangle \in \rho$ *and* $\langle q, \mathsf{e}, q' \rangle \in \alpha$ *as* $C : (m, q) \rightarrow q'$ *and* $C : q \xrightarrow{\mathsf{e}} q'$*, respectively, and letting* $q \rightarrow^* q'$ *represent transitive closure of* ρ *and* α*, the following holds:*

1. $\kappa(q^0(\texttt{this})) = \{\texttt{this}\}$ *(the object initially knows only its own identity);*
2. $q^0(\texttt{this}) \rightarrow^* q \implies \{\texttt{this}\} \in \kappa(q)$ *(the object always knows its own identity);*
3. $C : (m, q) \rightarrow q' \implies \kappa(q') \subseteq \kappa(q) \cup acq(m)$
   (the object knows another object through incoming method calls)
4. $C : q \xrightarrow{\mathsf{e}} q' \wedge \neg isMtd(msg(\mathsf{e})) \implies \kappa(q') \subseteq \kappa(q) \cup callee(\mathsf{e})$
   (or the object knows another object by creating it);

5. $C : (m, q) \to q' \implies isMtd(m)$
   (creation of objects is not part of the class semantics);
6. $C : q \xrightarrow{\mathsf{e}} q' \wedge isMtd(msg(\mathsf{e})) \implies \{callee(\mathsf{e})\} \cup acq(\mathsf{e}) \subseteq \kappa(q) \wedge \kappa(q') \subseteq \kappa(q)$
   (the callee and parameters of method calls made by the object must be known);
7. $C : q \xrightarrow{\mathsf{e}} q' \implies callee(\mathsf{e}) \neq \mathbf{this}$ *(no self calls are allowed).*

A class $C \in \mathbf{CL}$ is defined as a parameterized quintuple. Q is the set of states an object of class C can have. In general, Q can be an infinite set because an object may have fields that store object references. Furthermore, the object has an internal buffer to manage how incoming messages are processed. $q^0(\mathbf{this})$ represents the initial state the object has when it is created. The initial state is parameterized based on the identity of the object given when the class is instantiated. The transition relation ρ describes when the object receives a message and the effect of receiving that message on the object's state. ρ should be given such that any object of this class can only receive messages of certain forms, i.e., the class obeys a certain interface. In other words, any object of this class can only receive messages of certain forms. The transition relation α describes what an object does when it reaches a certain state, i.e., it can create another object or send a message to another object. The function κ describes which other objects an object knows when it is at some state. The single parameter represented by the variable **this** is assigned the value o when an object o of class C is created. When necessary, the members of the tuple are indexed with the class name for clarity.

For the class tuple to represent classes expressed by codes written in, e.g., ActJ, we put restrictions that explain the relationship between ρ, α and κ. The first four restrictions state how κ can evolve. Initially an object knows no other object but itself. An object can always refer to its own identity regardless of what kind of transition it previously has. It corresponds directly to the keyword **this** in ActJ. As in the programming language, there are two ways how an object o may know another object o', namely through references passed on as parameters and by creating o'. Once the reference o' is available to o, o can decide whether to store this information. As storing this information (through variable assignments) may cause o to forget another reference that is previously stored, the evolution κ is described using the subset relation. The freshness of a newly created object and its assignment of initial state are handled within the operational semantics, as hinted by the fifth restriction. Using this knowledge, we restrict the method calls produced by the object such that both the callee and the parameters of the method calls are known to the object. After making a method call, the object may forget some references to other objects. The seventh restriction on no self call acts as a simplification for easier classification of events into input or output events. Changes that occur with a self call can be simulated by making a nondeterministic choice of states as the effect of receiving or sending a message.

We assume that classes are defined such that their instances are input-enabled with respect to some consistent interface. The message part used in ρ and α is assumed to always be well-typed. The function $extMtd(C)$ extracts the interface supported by C. The function returns the set of method call messages that objects of class C can receive.

Example 1. The `Server` class can be represented as follows. Each state in $Q_{\mathtt{Server}}$ is partitioned into five parts: internal state $q_i \in \{\mathtt{r}, \mathtt{d}, \mathtt{p}\}$ representing `r`eady, `d`o, and

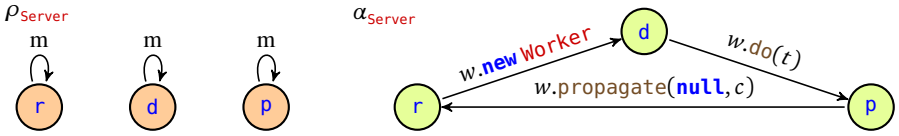


Fig. 4. Graphical representation of the message receive and action relations for `Server` class where $m = \texttt{serve}(c,t)$ and only the internal states are represented

propagate states, respectively; the client identity c; the worker identity w; the query y; and the internal message queue u. We use the sequence data structure $Seq\langle \mathbf{M} \rangle$ to represent the message queue. By virtue of the internal message queue and the infinite number of object identities, $Q_{\texttt{Server}}$ is an infinite set.

The initial state is $\langle \texttt{r}, \texttt{null}, \texttt{null}, \texttt{null}, [] \rangle$. The message receive relation $\rho_{\texttt{Server}}$ is represented symbolically by

$$\texttt{Server} : (\texttt{serve}(c', y'), \langle q_i, c, w, y, u \rangle) \rightarrow \langle q_i, c, w, y, u \cdot \texttt{serve}(c', y') \rangle \ .$$

Formally this representation is translated as the following relation:

$$\{\texttt{Server} : (m, q) \rightarrow q' \mid \exists c', y', q_i, c, w, y, u \bullet m = \texttt{serve}(c', y') \wedge q = \langle q_i, c, w, y, u \rangle \wedge q' = \langle q_i, c, w, y, u \cdot \texttt{serve}(c', y') \rangle\} \ .$$

Stated in words, the server can always receive a request, which is then stored in the internal message queue. Other parts of the server's state remain the same. Applying *extMtd* to `Server` results in

$$\{\texttt{serve}(c, y) \mid \exists c, y, q, q' \bullet \texttt{Server} : (\texttt{serve}(c, y), q) \rightarrow q'\} \ .$$

This set represents the interface `IServer` that the `Server` class is implementing.

The action relation $\alpha_{\texttt{Server}}$ is represented symbolically as follows.

A1. $\texttt{Server} : \langle \texttt{r}, \_, \_, \_, \texttt{serve}(c,t) \cdot u \rangle \xrightarrow{w.\texttt{new Worker}} \langle \texttt{d}, c, w, t, u \rangle$
A2. $\texttt{Server} : \langle \texttt{d}, c, w, t, u \rangle \xrightarrow{w.\texttt{do}(t)} \langle \texttt{p}, c, w, t, u \rangle$
A3. $\texttt{Server} : \langle \texttt{p}, c, w, t, u \rangle \xrightarrow{w.\texttt{propagate}(\texttt{null}, c)} \langle \texttt{r}, c, w, t, u \rangle$

An underscore represents any value and is used when the value of the particular part of the state is of no significance. The first relation states that the server is in the internal state of `r` where it is ready to process another `serve` message provided one is present in the internal message queue. As can be seen in Fig. 2, the server then creates a new worker. In this situation, it is important for the next steps that the server remembers the query, the client and worker references. The second and third relations proceed with the execution of the method `serve`, where `d` and `p` act similar to a program counter. All in all, this relation mimics the `serve` method of class `Server` given in Fig. 2. Graphically, $\rho_{\texttt{Server}}$ and $\alpha_{\texttt{Server}}$ can be viewed as in Fig. 4, by focusing only on the internal states.

The known object function $\kappa_{\texttt{Server}}$ is represented symbolically as follows.

K1. $\kappa(\langle \texttt{r}, \_, \_, \_, \_ \rangle) = \{\texttt{this}\}$
K2. $\kappa(\langle \texttt{d}, c, w, \_, \_ \rangle) = \{\texttt{this}, c, w\}$
K3. $\kappa(\langle \texttt{p}, c, w, \_, \_ \rangle) = \{\texttt{this}, c, w\}$

In r, the server forgets specific information about all previous requests that came in. When the server is processing a request, however, it is important to keep track of the client and worker references as explained previously. Thus in d and p, both the client and the worker references are tagged as known.

Discussion. Another way to describe the behavior of the objects is by assigning a behavior instance to each object. The change in behavior of object after receiving or producing a message is defined by a global relation (see, e.g., [61,4]). This approach has the drawback of giving less structure to play with. By having classes as templates, the behavior of objects can be defined without having to provide a global relation that works for all objects. See [13] for more comparison between object-based and class-based approaches on defining object's behavior.

Din et al. [20] introduce 4-event semantics to faithfully capture asynchronicity (i.e., allowing network delay between sending and receiving), similarly to the dual send and receive in CCS [41] and π-calculus [42]. In this semantics, the sending and the actual start of the method calls or creation are split into different event types. This distinction plays a role in the specification part, as it allows specifications to be represented locally at the cost of a more complex well-formedness condition on the generated traces.

The class definition above does not constrain an object to behave strictly as a pure concurrent object or a pure actor with multitasking capability. In particular, it does not enforce that the actions an object takes correspond to processing a single message. This means that the definition does not explicitly restrict the object to have a single-threaded computation, except that there cannot be two messages being sent out in parallel due to the interleaving nature of the transition relation. In this chapter, we assume that the action relation α follows an actor model with multitasking, such as ABS [33].

3.3 Operational Rules

To describe the interaction between a number of objects, we need run-time configurations that capture the current state of those objects. The current state of each object determines how objects act and react to incoming messages.

Definition 2 (Run-time configuration). *A* run-time configuration *of an object o is a tuple $\langle C, o, q\rangle$ where C is the class of o and $q \in Q_C$ is the run-time state of the object. The configuration of a* group *of objects $O \subseteq \mathbf{O}$ is a set of configurations $\mathcal{C}$ where for each object $o \in O$ there is at most one configuration of o in $\mathcal{C}$.*

A run-time configuration of an object o consists of its class C, its identity o and its current state q, with respect to the state description of its class. The identity of the object is used to replace **this** occurring in any part of the class tuple (i.e., o acts as the parameter in the class definition). For the remainder of the chapter, we use the notation $C\ o : q$ to represent a run-time configuration of o.

Example 2. $\langle \texttt{Server}, s, \langle \texttt{r}, \textbf{null}, \textbf{null}, \textbf{null}, \texttt{serve}(c_1, y_1) \cdot \texttt{serve}(c_2, y_2)\rangle\rangle$ is a possible configuration for a Server object s. This states that s has two incoming serve method calls and currently is in the initial state to process the next incoming request.

$$\textsc{ObjectCreation}\quad \frac{C_1 : q \xrightarrow{\mathsf{e}} q' \qquad \mathsf{e} = o_2.\mathbf{new}\ C_2 \qquad o_2\ \mathit{fresh}}{\{C_1\ o_1 : q\} \uplus \mathcal{C} \xrightarrow{o_1 \to o_2.\mathbf{new}\ C_2} \{C_1\ o_1 : q', C_2\ o_2 : q^0_{C_2}(o_2)\} \uplus \mathcal{C}}$$

$$\textsc{MessageSend}\quad \frac{C_1 : q_1 \xrightarrow{o_2.mtd(\overline{p})} q_1' \qquad C_2 : (mtd(\overline{p}), q_2) \to q_2'}{\{C_1\ o_1 : q_1, C_2\ o_2 : q_2\} \uplus \mathcal{C} \xrightarrow{o_1 \to o_2.mtd(\overline{p})} \{C_1\ o_1 : q_1', C_2\ o_2 : q_2'\} \uplus \mathcal{C}}$$

Fig. 5. Core operational semantics of concurrent object systems

The way the run-time configuration is affected by interaction between objects is shown in Fig. 5, representing the core operational semantics of concurrent object systems. The semantics consists of two rewrite rules that modify the configurations: ObjectCreation and MessageSend. In each of these rules, the transition from one configuration $\mathcal{C}$ to another $\mathcal{C}'$ is labeled by the corresponding event e that is represented by the rule. Textually, each transition has the form $\mathcal{C} \xrightarrow{e} \mathcal{C}'$. The disjoint union operator $\uplus$ ensures the consistency of having a single configuration of one object within C. When there are multiple ways to apply the rewrite rules, one is chosen at random.

ObjectCreation is applicable when an object o_1 is in a state q where it can create a new object of some class C_2 according to its class action relation. The result is that a new object o_2 with fresh identity is created and added into the configuration. The state of this new object is the initial state $q^0_{C_2}(o_2)$ of its class description.

MessageSend states the changes to two parties that act as end points of a method call (i.e., the caller and the callee). An object o_1 (the caller) can perform an asynchronous method call $mtd(\overline{p})$ on another object o_2 (the callee) if the following conditions hold. First, the caller is in a state where it can send a message $mtd(\overline{p})$ to the callee. By the condition placed on the class definitions, the caller knows the callee. Second, the callee is able to receive that message. With the input-enabledness assumption in place, this part is always fulfilled. Thus, the resulting states q_1' and q_2' of both the caller and the callee, respectively, can be derived when the method call is made.

A straightforward consequence of the operational semantics is that an object always keeps its class designation as shown by the lemma below.

Lemma 1 (Class preservation). *An object never changes its class.*

Proof. Follows from Def. 2 and the operational semantics rules.

Being able to refer to specific entities with regards to the behavior of objects is useful later on to structure the behavior of a group of objects without needing extra constructs. Class preservation of objects serves as an important basis to this usage.

Discussion. The operational semantics above has only two rules. The rules cover the necessary observable operations on a concurrent object model (cf. [58]). The internal computation needed to produce the messages is abstracted away in our model by means of object states. Vasconcelos and Tokoro [61] even reduced the number of kinds of observable operations into one, by just considering the method calls (which in their work

are simply called *messages*). Object creation is encoded implicitly within the internal computation. In a single transition, the number of objects that are present in the configuration can change. In our case, the object creation needs to be considered separately to extract the initial state from the class definition.

The MESSAGESEND rule pairs the action relation of one class to the message receive relation of another class. Between pairs of objects o and o', the order of how methods are called by o to o' is the same as the order of the same calls received by o'. The actor model [16,3,4] retains pure asynchronicity, meaning that it allows message overtaking even between pairs of objects. Yonezawa, Briot and Shibayama [63] argued, however, that having this guarantee eases describing distributed algorithms.

As each object can concurrently perform their internal processing at possibly different speeds, there is a need for external nondeterminism. This need is covered by the random application of operational rules. This choice raises the question about the fairness of choosing the objects to which the rules are applied. As this issue is not prominent for our discussion on specification and verification (unlike in the discussion on actors [4], for example), we assume weak fairness on the operational rule application.

3.4 Translation to Traces

Given a configuration, we can define an execution from this configuration as a sequence of interleaved configurations and configuration transition, where each transition represents an application of the operational semantics rules. We can extract from an execution the trace by taking the concatenation of the labels of configuration transitions. If we know the default initial configuration, the traces can be used to abstract from the internal representation of the individual objects [30,12]. For this reason, we use traces as semantic foundation for specifications.

The following definition states what we mean by executions and traces. The symbol $\surd$ distinguishes a *maximal* execution (i.e., a finished execution where no more operational rule can be applied [8, p. 96]) from an execution that can still make progress.

Definition 3 (Execution and trace). *An* execution *is a sequence of interleaved configurations and events* $\mathcal{C}_0 \xrightarrow{e_1} \mathcal{C}_1 \xrightarrow{e_2} \mathcal{C}_2 \cdot \ldots$ *which may end with* $\surd$ *after a configuration. A* trace $t \in Seq\langle \mathbf{E} \cup \{\surd\}\rangle$ *of an execution is the projection of the execution to the events by leaving out the configurations. The trace* t *ends with* $\surd$ *if the corresponding execution ends with* $\surd$.

Example 3. Let a be some `Main` object which represents the initial object whose task is to set up the server system. Assume as well a `Client` object c has been created. Using an underscore to represent a configuration content of no interest, the following execution describes a possible way how the server is created and how it responds to an incoming request from the client.

$$\{\texttt{Main}\ a : \_, \texttt{Client}\ c : \_\} \xrightarrow{a \to s.\texttt{new Server}}$$
$$\{\texttt{Main}\ a : \_, \texttt{Client}\ c : \_, \texttt{Server}\ s : \langle \texttt{r}, \_, \_, \_, [] \rangle\} \xrightarrow{a \to c.\texttt{link}(s)}$$
$$\{\texttt{Main}\ a : \_, \texttt{Client}\ c : \_, \texttt{Server}\ s : \langle \texttt{r}, \_, \_, \_, [] \rangle\} \xrightarrow{c \to s.\texttt{serve}(c,y)}$$
$$\{\texttt{Main}\ a : \_, \texttt{Client}\ c : \_, \texttt{Server}\ s : \langle \texttt{r}, \_, \_, \_, \texttt{serve}(c,y) \rangle\} \xrightarrow{s \to w.\texttt{new Worker}}$$

$$\{\texttt{Main}\ a : \_, \texttt{Client}\ c : \_, \texttt{Server}\ s : \langle \texttt{d}, c, w, y, [] \rangle, \texttt{Worker}\ w : \_\} \xrightarrow{s \to w.\texttt{do}(y)}$$
$$\{\texttt{Main}\ a : \_, \texttt{Client}\ c : \_, \texttt{Server}\ s : \langle \texttt{p}, c, w, y, [] \rangle, \texttt{Worker}\ w : \_\} \xrightarrow{s \to w.\texttt{propagate}(\texttt{null}, c)}$$
$$\{\texttt{Main}\ a : \_, \texttt{Client}\ c : \_, \texttt{Server}\ s : \langle \texttt{r}, \_, \_, \_, [] \rangle, \texttt{Worker}\ w : \_\}$$

This execution shows six transitions. The first two transitions complete the setup of the setting, by having the server object created and having it linked to the client. The ObjectCreation and MessageSend rules are respectively applied by assuming that the states of the main and client objects are the state where they are applicable. The third transition states that the client sends a request to the server. The server stores this request in its internal queue. Then the server processes this request in the remaining transitions as described by the server action relation given in Ex. 11. First it creates a new worker and takes out the `serve` message out of the its queue. As an effect of the transition, the new worker reference is stored as well as the client reference and the query from the `serve` message. The internal state also changes to `do`. Then, it initiates the result propagation to this new worker. The server is then back to the state where it is ready to process another request (although no more pending requests are present). In the last two transitions, we assume that the worker state changes accordingly.

The following trace can be extracted from the execution.

$$a \to s.\texttt{new}\ \texttt{Server} \cdot a \to c.\texttt{link}(s) \cdot c \to s.\texttt{serve}(c, y) \cdot s \to w.\texttt{new}\ \texttt{Worker} \cdot s \to w.\texttt{do}(y) \cdot s \to w.\texttt{propagate}(\texttt{null}, c)$$

This trace is not maximal, because the operational rules can still be applied to the worker object for processing the query.

To obtain specific information related to a certain object o from a trace t, we use the projection operator $t{\downarrow_o}$. This operator states the projection of t to o where all events where o is neither caller nor callee are removed from t. When necessary, the object parameter can be enriched with *callee* or *caller* to denote that we are focusing on the events where o is the callee or the caller, respectively. This operator is naturally lifted to a trace set T and a set of objects O. Other operators are introduced as needed.

Example 4. Let t be the trace from the previous example. Then if we project t to the server object s, then we obtain the following trace.

$$t{\downarrow_s} = a \to s.\texttt{new}\ \texttt{Server} \cdot c \to s.\texttt{serve}(c, y) \cdot s \to w.\texttt{new}\ \texttt{Worker} \cdot s \to w.\texttt{do}(y) \cdot s \to w.\texttt{propagate}(\texttt{null}, c)$$

An important restriction on the class definition (Def. 1, No. 6.) is that a method call can be made only if the objects in the parameters of the method call are known. This restriction can be transferred to traces by requiring that an object can only send messages to other objects it has been exposed to. A trace contains enough information to determine whether an object is exposed to another object. This information is extracted using the functions *acq* and *cr*. The function $cr(b.\texttt{new}\ C)$, short for *created*, extracts the identity of the newly created object from an object creation (i.e., the callee b). These functions are lifted to events and traces.

Definition 4 (Well-formed trace). *Let e be a method call event* $o \to o'.mtd(\overline{p})$. *A trace* t *is* well-formed *if*

$$\forall o \in \mathbf{O}, t' \cdot e \in Pref(t) \bullet \{o'\} \cup acq(e) \subseteq acq(t'{\downarrow_{o,callee}}) \cup cr(t'{\downarrow_{o,caller}}) .$$

The definition above states that for every method call an object makes, it must know the identity of the object it is calling and also the identities of each object present as parameters of the method call. These identities are collected from previous method calls the caller receives (through the acquaintance function) and the objects that have been created directly by the caller (represented by the created function). Other properties of trace well-formedness, such as the freshness of the newly created objects, can be defined in a similar way. We leave their definition to the reader as an exercise[4]. Because there is no self-call and an object only knows its own identity when it is created as per restriction on κ (Def. 1), we obtain the following corollary.

Corollary 1. *A non-empty well-formed trace t begins with a creation event.*

The following lemma shows that when we start from a configuration containing an object in its initial state, using the core operational rules and the restrictions given on the κ function (Def. 1), the generated trace(s) is well-formed.

Lemma 2 (Well-formedness preservation). *Let* **CL** *be a set of classes and the singleton $\mathcal{C} = \{C\ o : q_C^0(o)\}$ the initial configuration of object o of class $C \in$* **CL***. Then, by applying the operational rules from Fig. 5, the generated trace is well-formed.*

Proof (sketch). Two important factors for the proof are the monotonicity of the functions *acq* and *cr*, and that no object identities absent from the events are introduced by these functions. Because these factors are weaker than the restrictions on κ in Def. 1, the generated trace is well-formed.

Discussion. We have introduced two different semantics for our objects: the operational semantics and the trace semantics. These semantics are chosen because they represent the two common abstraction levels used for reasoning. The operational semantics allows a program to be executed with respect to some configuration. Trace semantics typically exhibits full abstractness quality as shown in various concurrent models, e.g., [35,2,44]. It is attractive as the basis for specifying the functional property of a system, because the specification can independently be given without determining how it is implemented.

Talcott [58] provided more layers of composable semantics of actors to allow local reasoning at different levels of abstractions. Starting with an operational semantics similar to ours (Sect. 3.3), other layers are formed by hiding external events belonging to the environment, focusing only on the partial order between events, and hiding internal events. Her approach relies on retaining possible global timings of each event.

4 Systems and Components

Having a semantics for a group of objects is the basis for understanding how a system behaves. As motivated in Sect. 1, a natural way to structure the grouping is to use classes to form components. A system's behavior can be understood by composing the behavior of its components. In this section, we explore the possible ways to structure groups of objects into components and link them with the notion of open systems.

[4] For example, Din et al. [20] provides such a definition.

4.1 Closed Systems

First, we need to know whether we have enough information to apply the rules of the operational semantics. This means we need to have the necessary class definitions.

Definition 5 (Definition-complete). *Let* $\mathbf{C} \subseteq \mathbf{CL}$ *be a set of classes.* $\mathbf{C}$ *is* definition-complete *if for each method call* $o.mtd(\overline{p})$ *such that* $class(o) = C'$ *and object creation* `new` C' *appearing in* α_C *of any class* $C \in \mathbf{C}$*, it holds that* $C' \in \mathbf{C}$*.*

A set of classes is definition-complete if for each object present within the interaction, the class definition of that object is also available within the set.

Definition 6 (Closed system). *A* closed system $CS = (\mathbf{C}, C_0)$ *is a definition-complete set of classes* $\mathbf{C}$ *with a distinguished* activator class $C_0 \in \mathbf{C}$*.*

A closed system contains all the class definitions necessary for knowing precisely how each object within the system behaves. An activator class is the class of the initial object of the system (similar to starting a Java program, for example, with some initial class containing a `main` method). This initial object should then create other objects necessary for the system to run.

Definition 7 (Trace semantics of closed systems). *The trace semantics of a closed system* $CS = (\mathbf{C}, C_0)$ *is the trace set* $Traces(CS)$ *where for each* $t \in Traces(CS)$*, there is an execution starting from an initial configuration* $\mathcal{C} = \{C_0\ o : q^0_{C_0}(o)\}$ *whose trace is* t*.*

The trace semantics of a closed system is a trace set containing all traces that can occur from a singleton initial configuration of an object of the activator class. Using the closed system definition, we can define the trace semantics of a class.

Definition 8 (Trace semantics of classes). *The trace semantics of a class* C *is the trace set* $Traces(C)$ *where for each trace* $t \in Traces(C)$*, there is a closed system* $CS = (\mathbf{C}, C_0)$ *such that* $C \in \mathbf{C}$ *and there is a trace* $t' \in Traces(CS)$ *such that* $t = t'{\downarrow_o}$ *where* $class(o) = C$*.*

The trace semantics of a class C is obtained by taking all traces of all closed systems that contain C, then projecting all those traces down to objects of class C.

Example 5. A trace of the `Server` class is $a \rightarrow s.\texttt{new Server} \cdot c \rightarrow s.\texttt{serve}(c,t) \cdot s \rightarrow w.\texttt{new Worker} \cdot s \rightarrow w.\texttt{do}(t) \cdot s \rightarrow w.\texttt{propagate}(\texttt{null}, c)$. This is obtained by taking the projection of the trace from Ex. 3 to the `Server` object s.

Discussion. The trace semantics can as well be directly defined based on the class definition. However, in a concurrent nondeterministic setting, it is not easy to do. In particular all possible message reception and sending has to be taken into account. Ahrendt and Dylla [5], for example, apply *guess and merge* approach when synchronizing the execution of different method calls to obtain the trace set.

4.2 Open Systems and Components

A closed system deals with a group of objects that are completely executable without any influence from outside, while single classes only deal with the behavior of each of those objects. A gap is present between single classes and closed systems when we focus our attention only on the behavior of some particular group of objects of a closed system. When this group expands dynamically as the system continues to execute (e.g., the workers of the server that handle a request in our running example), it is impractical to know how this group as a whole behaves by referring to the behavior of each of the objects in the group. In turn, it is impractical to verify whether the behavior of a closed system follows directly from the class descriptions contained in that system.

To help reason about the behavior of closed systems from single classes, we abstract a collection of single classes into *components*. Components should share the characteristics of closed systems and single classes and allow hiding internal behavior.

In this chapter, we choose an abstraction based on object creation. This abstraction makes it possible to refer to a component by the class of the component's initial object. Its advantage lies in the inference how the component is structured. That is, the structure of a component comes with how the classes behave, instead of needing to state individually which classes and subcomponents are contained in the component.

Definition 9 (Creation-complete). *Let* $\mathbf{C} \subseteq \mathbf{CL}$ *be a set of classes.* $\mathbf{C}$ *is* creation-complete *if* $C' \in \mathbf{C}$ *for each object creation* `new` C' *appearing in* α_C *of any class* $C \in \mathbf{C}$.

A set of classes is creation-complete if for each created object, its class is within that set. In comparison to definition-complete (Def. 5), creation-complete allows the possibility of having the behavior of some objects unknown. In the context of the interaction of a group of objects, the creation-complete notion allows the behavior of some objects whose references are passed on within the interaction to be left *open*.

Definition 10 (Component). *A* Component $\mathbb{C} = (\mathbf{C}, C_0)$ *is a creation-complete set of classes* $\mathbf{C} \subseteq \mathbf{CL}$ *with some activator class* $C_0 \in \mathbf{C}$.

A component is essentially a system that starts with an object of some activator class. In comparison to a closed system, a component may have some parts *open* to be matched with some context. A set only consisting of a single class is possibly not a component because it may create an object of a different class.

Example 6. The pair ({`Server`}, `Server`) is not a component because a server may create a worker, and the class `Worker` is not within the set of classes. The pair ({`Worker`}, `Worker`) on the other hand is a component because the only objects a worker may create are of the same class. In addition, we can combine the `Worker` component with the `Server` class to create a new component: ({`Server`, `Worker`}, `Server`). Note that none of these components are closed systems because they are missing the `Client` class.

Definition 11 (Context). *A context of a component* $(\mathbf{C}, C_0)$ *is* $\mathbb{X} = (\mathbf{C}^x, C_0^x)$ *such that* $\mathbf{C}^x \cup \mathbf{C}$ *is definition-complete and* $C_0^x \in \mathbf{C}^x$.

A context is a set of classes with some activator class that completes the class definition of a component. A context may use the same classes as the component, so the

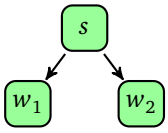
component can have certain expectation about the behavior of the context. But since we are following the programming to interfaces principle (see Sect. 2.1), the component generally does not know how objects of the context behave other than they adhere to certain interfaces. A context on its own does not need to be definition-complete or creation-complete. Paired with a matching component, we get a closed system.

Example 7. Let `Main` be a class whose instances create a client and a server and link the server to the client. Then, the pair $(\{\texttt{Client}, \texttt{Main}\}, \texttt{Main})$ is a context of the component $(\{\texttt{Server}, \texttt{Worker}\}, \texttt{Server})$.

Similar to the trace semantics of closed systems and classes, the behavior of a component can be defined by taking all traces produced by the combination of all possible contexts with the component. The main issue is to which objects should the resulting traces be projected. These objects can be derived by following an object creation tree whose root is the initial object of the component. The object creation tree keeps track of objects transitively created by the initial object of the component. Instead of formally defining the tree, we illustrate it using the following example.

Example 8. Consider the server component $(\{\texttt{Server}, \texttt{Worker}\}, \texttt{Server})$ given in Ex. 6. The following trace illustrates the creation of the server object s and its (incomplete) reaction to two requests. All creation events are highlighted.

$$a \rightarrow s.\texttt{new Server} \cdot a \rightarrow c.\texttt{link}(s) \cdot c \rightarrow s.\texttt{serve}(c, t_1) \cdot$$
$$s \rightarrow w_1.\texttt{new Worker} \cdot s \rightarrow w_1.\texttt{do}(t_1) \cdot s \rightarrow w_1.\texttt{propagate}(\texttt{null}, c) \cdot$$
$$c \rightarrow s.\texttt{serve}(c, t_2) \cdot s \rightarrow w_2.\texttt{new Worker}$$



Each time the server processes a request from the client, it creates a worker. Taking s as the root of the tree, the tree has also nodes containing the workers w_1 and w_2. Graphically, this tree is shown to the right of the trace.

By virtue of the object creation tree, the objects of the component with respect to some well-formed trace can be tracked using the *identity extractor* function.

Definition 12 (Identity extraction). *Let t be a trace and C is some (activator) class. The* object identity extractor *function $idx(t, C)$ is defined as follows.*

$$idx([\,], C) = \emptyset,$$
$$idx(t \cdot e, C) = \begin{cases} \{callee(e)\} & \text{if } msg(e) = \texttt{new } C \wedge idx(t, C) = \emptyset \\ idx(t, C) \cup \{callee(e)\} & \text{if } msg(e) = \texttt{new } C' \wedge caller(e) \in idx(t, C) \\ idx(t, C) & \text{otherwise} \end{cases}$$

The function idx extracts the identities of objects directly and indirectly created by the initial object of the component. The function takes as arguments a trace t and a class C. It works by determining the initial instance o of C in the trace. This initial instance is the root of the object creation tree. Then the function recursively collects all the objects transitively created by o.

In a trace, there can be more than a single instance of C being created. For defining the behavior of a component, only the first instance of C and other instances transitively

created from that first instance are returned by *idx*. This restriction ensures that an object creation tree can be constructed from the result of applying *idx*. We lift *idx* to a trace set T by taking the union of the application of *idx* to each trace in T.

Example 9. Let t be the trace given in Ex. 8. Then, $idx(t, \texttt{Server}) = \{s, w_1, w_2\}$.

Using the object identity extractor function, we are ready to define the first variant of the component trace semantics. We call this variant as the *plain* trace semantics of a component. The term plain refers to the lack of hiding performed on the internal interaction between objects within the component.

Definition 13 (Plain trace semantics of components). *Let* $\mathbb{C} = (\mathbf{C}, C_0)$ *be a component. The* plain trace semantics *of* $\mathbb{C}$ *is a trace set* $Traces(\mathbb{C})$ *where for each trace* $t \in Traces(\mathbb{C})$*, there is a context* $\mathbb{X} = (\mathbf{C}^x, C_0^x)$ *of* $\mathbb{C}$ *such that in the resulting closed system* $CS = (\mathbf{C}^x \cup \mathbf{C}, C_0^x)$*, there is a trace in* $t' \in Traces(CS)$ *and* $t = t'{\downarrow_{idx(t', C_0)}}$.

Implicit within the definition is the usage of object creation tree to build the component instances. Discussion on other ways to define the component instances is deferred to the end of this section. The closed system used to obtain the traces is just one way to compose the component with the context. The activator class of the resulting closed system is taken from the context because it is the context which decides when the component is instantiated.

This trace semantics supplies enough information to link the operational semantics to an openness property of the trace semantics. This property is crucial for proving the soundness of the proof system presented in Sect. 5. The openness property shows that once an object of the component is exposed to the context (i.e., the component's environment), the context can do anything with it, in particular making all possible method calls (with respect to all other exposed objects). We adopt the notation from [37], where the objects of the component instance are grouped into L (for *l*ocal) and the objects of the context (i.e., objects not part of the component instance) are grouped into F (for *f*oreign). The local objects can be extracted from the plain trace set of a component by applying the identity extraction function. For defining the openness property, the exact content of L is left open.

Definition 14 (Open system trace sets [37]). *Let* T *be a trace set of a group of objects* L*,* $F = \mathbf{O} - L$ *and* $e = o \rightarrow o'.mtd(\overline{p})$ *an event such that* $o \in F$*,* $o' \in L$*, and* $mtd(\overline{p}) \in extMtd(class(o'))$*.* T *is* open *if*

$$\forall t \in T, e \in \mathbf{E} \bullet \{o'\} \cup acq(e) \subseteq exposed(t, F) \cup F \implies t \cdot e \in T$$

where $exposed(t, F) = cr(t{\downarrow_{F,caller}}) \cup acq(t{\downarrow_{F,callee}})$.

We call a trace set *open* with respect to a set of local objects if for each trace in the trace set, a method call event directed to an exposed local object can be constructed using the exposed local objects and the context objects. Appending that method call to the end of the trace results in another trace which is contained in the trace set. The exposed references are derived by taking all local objects created by the context and all local objects whose references are passed on to the context through method calls.

To instantiate this property in our trace semantics we assume without loss of generality that the identity of the initial object of the component is fixed, and take the collection of all objects created by that initial object in all traces. The following lemma shows the connection between the core operational semantics presented in this chapter and the trace semantics, with respect to the openness property.

Lemma 3 (Openness of plain trace semantics of components). *If* $\mathbb{C} = (\mathbf{C}, C_0)$ *be a component, then* $Traces(\mathbb{C})$ *is open.*

The main idea behind the proof is that for each context that produces a trace of that component, we can always construct another context such that the any exposed object is immediately used by the context in all possible ways. Thus, the other context ensures that the open system trace property holds.

Proof. Let $t \in Traces(\mathbb{C})$ be achieved by composing $\mathbb{C}$ with some context $\mathbb{X} = (\mathbf{C}^x, C_0^x)$. Let also $L = idx(Traces(\mathbb{C}, C_0))$, $F = \mathbf{O} - L$, and $o' \in exposed(t, F) \cap L$. Because of the interface model[5], we can assume without loss of generality that

- $\mathbf{C}^x \cap \mathbf{C} = \emptyset$, and
- C_0^x is such that all necessary objects of the context are created before the activator class of the component is created.

We create another context $\mathbb{X}' = (\mathbf{C}^{x\prime}, C_0^{x\prime})$ that is the same as $\mathbb{X}$, except that for every class $C \in \mathbf{C}^x$ we create C' where we pick some new method name mtd' that does not appear in any class and add as many states as necessary such that

1. for all $o \in \kappa_{C'}(q), o'' \in F$, $C' : q \xrightarrow{o''.mtd'(o)} q'$
   (the knowledge is shared among all objects of the context),
2. $C' : (mtd'(o), q) \to q'$ *(all objects of the context remain input-enabled)*, and
3. if $o' \in \kappa_{C'}(q)$, then for all $mtd \in extMtd(class(o'))$,
   $C' : q \xrightarrow{o'.mtd(\overline{p})} q'$ such that $acq(\overline{p}) \subseteq \kappa_{C'}(q)$
   (all objects of the context can send to any exposed object of the run-time component a method call within the implicit interface of the class of that exposed object).

It can be checked that C' follows the class definition (Def. 1) and $\mathbb{X}'$ remains a context (Def. 11). By the operational rules (Sect. 3.3) and Def. 13, we can obtain the projected trace t where all objects of the context share the identities of all objects of the context and exposed objects of the component. Because of point 3, once o' is exposed any object of the context $o \in F$ can make a method call to o'. By the input-enabledness assumption, MESSAGESEND can be applied and we get a projected trace $t \cdot o \to o'.mtd(\overline{p}) \in Traces(\mathbb{C})$. □

[5] Not following the programming to interfaces principle brings a problem called *replay* [56]. The replay problem appears when the component exposes some object to an object of the context whose class is part of the component. Because the implementation of the object is fixed, that context object can only use the exposed component object in a specific way. In particular, this context object may only store the exposed component object and openness is not achieved.

As stated earlier, we want to bridge classes and closed systems. This goal is achieved by hiding the *internal* behavior that happens between objects within the component. Such a view allows bottom-up reuse of the component, for example, when it is a library or a framework. Moreover, the view is suitable to deal with open systems, in particular systems with a non-software environment (e.g., GUI interaction with a human user – see [48] for more discussion). By using *idx*, hiding is straightforward to define.

Definition 15 (Component trace semantics). *Let* $\mathbb{C} = (\mathbf{C}, C_0)$ *be a component and* $Traces(\mathbb{C})$ *its plain trace semantics, so that* $L = idx(Traces(\mathbb{C}), C_0)$ *is the set of local objects of* $\mathbb{C}$*. The* component trace semantics*, written* $Traces([C_0])$*, is the trace set* $Traces(\mathbb{C})\downarrow_{\mathbf{O}-L}$.

To distinguish the plain trace semantics of components (Def. 13) from the one where hiding is performed, we use the notation $[C_0]$. This notation, read as boxed C_0, comes from the way we structure our component instances. A component instance can be thought of as a box of objects starting from the initial instance of the activator class C_0. By projecting the traces to the set of objects of the context, the above definition only reflects the *observable* behavior representing the interaction with the context. The component's users only need to focus on the component's observable behavior. For example, the client is only interested in how the server component (as a whole) interacts with it, not in how the server does its job. As the projection does not change the interaction that happens on the boundary of the component, the following corollary holds.

Corollary 2. *If* $(\mathbf{C}, C_0)$ *is a component, then* $Traces([C_0])$ *is open.*

Also of interest is that any trace in a component trace semantics contains at most a single creation event. This event represents the creation of the initial object of the activator class. In fact, this event is always the first event of a non-empty trace.

Lemma 4 (Initial creation event). *Let* $\mathbb{C} = (\mathbf{C}, C_0)$ *and* $Traces([C_0])$ *its component trace semantics. The following properties hold.*

1. $\forall e \cdot t \in Traces([C_0]) \bullet msg(e) = \texttt{new}\ C_0$
   (trace begins with creation of an instance of C_0*)*
2. $\forall e \cdot t \cdot e' \in Traces([C_0]) \bullet isMtd(msg(e'))$
   (no other creation event appears in the trace)

Proof. 1. We assume that initially only an object of the context exists. To instantiate the component, the initial object of the activator class must be created at some point. Due to the projection, the first event in any non-empty trace of the component trace semantics is creation of an instance of the activator class.
2. All other objects within the run-time component are created transitively by the initial object of the run-time component. These creation events are projected away. □

Discussion. The term *activator class* used in this chapter comes from OSGi [60]. In OSGi model, a component is instantiated by creating an object of a class that implements the `BundleActivator` interface, an interface each component is required to implement. The model where the component is instantiated by creating an object of the

activator class is not uncommon. For example, it coincides with the object adaptor of the CORBA Component Model [46] and the class factory of COM [40]. Should a need arise for having more than one initial object, it can be simulated by our model by having the activator class as a stub that only creates the other objects.

The main question with defining the behavior of components is how we separate the internal and observable behavior. For this purpose, the notions of *boundary* enclosing the objects of a component at run-time and encapsulation are important. With our component model, we identify three particular ways to group objects into instances of a component or run-time component. Following the component definition (Def. 10), we identify three particular ways to group objects into instances of a component, called *run-time components*: static, programmer-defined and dynamic. The division between static and dynamic stems from how the objects created during execution are organized within the component instances.

Static Run-Time Component. Given a component $(\mathbf{C}, C_0)$, its static run-time representation contains every object of whose class is in $\mathbf{C}$. As such, grouping the objects into the component is trivial to define, by following the class of each object. However, the drawbacks of doing so are numerous. As the run-time component contains all objects whose class in $\mathbf{C}$, every object is at the boundary. By having all objects at the boundary, the interactions between these objects are visible. In addition, components cannot share classes, as there is no way to separate the run-time instances of intersecting components. In our example, a static run-time component of the $(\{\texttt{Server},\texttt{Worker}\},\texttt{Server})$ component includes all server and worker objects. Hence, we cannot focus only on a single server with the workers it creates to represent the run-time view of the component. Not only the focus on the activator class is lost, it is also difficult to specify the desired behavior of the component.

Programmer-Defined Run-Time Component. A programmer-defined run-time component contains all objects in the way how the programmer defines it by specifying at the point of creation to which run-time component the newly created object belongs to. For example, in CoBox [53], this is done by extending the `new` statement with `in` o to say that the newly created object resides in the same run-time component as o. Various other ownership approaches can also be used (see [15] for a survey). This approach is the most flexible as it provides fine-grained information which objects are at the boundary, but also more complex to handle.

The task of defining the exact nature of the run-time components can also be deferred to the specification part. For example, Roth [50, Chap. 4] defines a *Reach* predicate to indicate whether some object o is reachable from o' through field and array accesses (although recall that concurrent objects are not allowed to directly access fields of other objects). While staying on the specification level, this approach offers more precision in stating the current content of some run-time component (i.e., allowing to state that some component may forget a reference).

Dynamic Run-Time Component. A dynamic run-time component contains all objects that are created directly or indirectly by the initial object of the activator class. In other words, the run-time component is formed from the object creation tree, with the initial object of the activator class as the root. This gives a more fine-grained grouping than the static approach, but is less specific than the programmer-defined

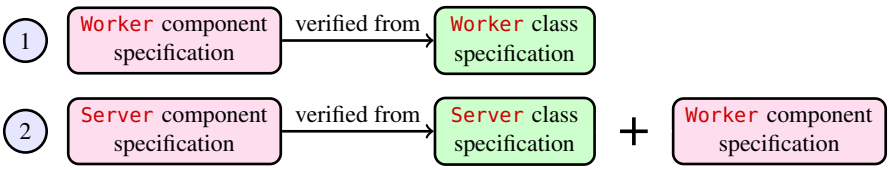


Fig. 6. Verification flow of the `Server` component

approach. Additionally, we can keep track of which objects are then on the boundary, while keeping track which new objects are included in the run-time component. This enables staying on focus on the behavior at the component boundary. After all, the hidden objects do not appear at the boundary and hence hiding them reduces the communication with the context. Because of its dynamicity, it is less straight forward than the static approach to give up front the exact instances of a component. In this chapter, we follow the dynamic approach of identifying run-time components which allows the use of the activator class to represent the component.

Our notion of context is similar to the notion of program closure in [50, Chap. 2], where it is defined as the minimal set of additional class skeletons needed for a set of classes to compile. The closure is used to create an observer that checks whether method call invocations maintain the program contract. In our setting, it is not necessary for the context to be the minimal set because traces involving external events are filtered out by the projection. That is, the form of this closure is not important.

5 Specification and Verification

Using the semantics of object classes and run-time component characterization based on the object creation tree, we now specify and verify the functional behavior of the components. As stated in the introduction, the main goal is to achieve verification of component specifications from the specifications of the class specifications. Component specifications that have been verified can be used to verify higher level components.

In terms of our running example, this means we give specifications of the functional behavior of `Server` and `Worker` classes and of the `Server` and `Worker` components. In this chapter, we assume the `Server` and `Worker` class specifications to hold, so we can use them directly in our verification effort. To give a complete verification, the `Server` and `Worker` class implementations given in Sect. 2 must be verified against their respective class specifications. As the main goal is to prove the `Server` component property, we may proceed the verification in two steps, as illustrated by Fig. 6. First, we verify the `Worker` component specification from the `Worker` class specification. Then, we use the verified `Worker` component specification together with the `Server` class specification to show that the `Server` component specification holds.

In this section, we describe a specification and verification technique to illustrate how this goal can be achieved. We provide the specifications for the classes and the

components, but we only illustrate the second part of the verification effort (i.e., proving the server component specification holds)[6].

5.1 Specification

An ideal specification technique should have a similar way to specify the behavior of classes and components. Our idea is to generalize the specifications of methods in the sequential case, which relate pre- to poststates, by relating input traces to output traces. To realize this idea, we use a specification format similar to the Hoare triple [29]. That is, the specification is of a triple form $\{p\}\ D\ \{q\}$, where p and q are *trace assertions* and D is either a class or a component. Informally, this triple means that if an input trace of D satisfies p, the output trace will satisfy q. In the following, we formally define the meaning of each part of this specification.

A trace assertion is a first-order logic formula in which the special *trace variable* \$ can be used. The trace variable represents the *caller suppressed* trace. This suppression, which yields an event core sequence, reflects the lack of knowledge on the receiver side (i.e., the entity we are specifying) who the sender of a message is. To achieve this suppression, we let the function *sup* transform a trace into caller suppressed trace. Wherever clear, we use caller suppressed trace and normal trace interchangeably.

Definition 16 (Trace assertion). *Let* \$ *be a trace variable representing a trace.* Trace assertions p, q *are defined inductively by the following first-order logic clauses:*

- *Expressions of boolean type are assertions (*\$ *may be present).*
- *If* p, q *are assertions and* $\underline{x}$ *is a variable, then* $\neg p, p \wedge q, \exists \underline{x} : p$ *are also assertions.*

Other logical operators, e.g., $\vee$, $\Longrightarrow$ and $\forall$, are derived in the usual way. For a trace assertion p, the function $free(p)$ returns the set of all free variables appearing in p.

To define the semantics of a trace assertion, we substitute all occurrences of the trace variable with the actual trace. As a generic substitution mechanism, we use the notation $p[\underline{x}/r]$ to denote the substitution of all (free) occurrences of a variable $\underline{x}$ or the trace variable by some expression or assertion r in a trace assertion p. We assume that all variables and all substitutions are correctly typed. Using first-order logic, we map the assertion to boolean values {**true**, **false**}. Given a trace t, we write $\vDash_{FOL} p[\$/sup(t)]$ if it is mapped to **true**, and $\vDash_{FOL} p$ if for any trace t, $\vDash_{FOL} p[\$/sup(t)]$. The *FOL* index indicates that the variable assignment is done using the first-order logic semantics.

Example 10. $\$ = \langle \underline{\text{this}}.\textbf{new}\ \texttt{Worker} \cdot \underline{\text{this}}.\texttt{do}(\underline{\text{y}}) \cdot \underline{\text{this}}.\texttt{propagate}(\underline{\text{v}}, \underline{\text{c}}) \rangle$ is a trace assertion stating that the trace starts with a creation of a `Worker` object, stored into the free variable this. Then, the worker is sent a query computation request followed by a result propagation. Only this sequence appears in the trace.

Using trace assertions as the foundation, a specification triple $\{p\}\ D\ \{q\}$ is described as follows. The triple specifies the output trace the instance of class or component D produces when faced with an input trace satisfying p. Because of their nature, we call p and q *input* and *output trace assertions*, respectively. All variables appearing only in q

[6] Interested readers can attempt the first part. A complete proof is available in [37].

(possibly due to an explicit creation of another object or an implicit exposure of locally created objects) should be existentially quantified. As convention, the initial object is referred to by the variable this. The precise trace semantics of the entity represented by D is as follows. For each trace $t \in \mathit{Traces}(D)$ whose input part satisfies p, its output part satisfies also q. This specification technique does not give information about the rest of the traces that do not satisfy p. Despite the underspecification, the specification triple eliminates traces which satisfy p and do not satisfy q.

To define the triple semantics, a trace t needs to be split into input and output traces. For that we need the set of objects L that represents entity D at run-time. The function $\mathit{split}(t, L) = (t{\downarrow_{F,caller}}{\downarrow_{L,callee}}, t{\downarrow_{L,caller}}{\downarrow_{F,callee}})$ does exactly so, where $F = \mathbf{O} - L$. The first part produces the input trace of t by focusing on events in t where the caller is a foreign object and the callee is the local object. The second part produces the output trace of t in a similar way.

In the case where the entity represented by D is a class C, L is taken to be a singleton object o whose class is C. We call $\{p\}\ C\ \{q\}$ a *class triple*. The *split* function ensures that in the input and output traces that are being considered in the semantics of the specification triple, only the interaction done by o appears.

Definition 17 (Class triple semantics). *Let C be a class and o an object such that $\mathit{class}(o) = C$. $\mathit{Traces}(C)$ satisfies $\{p\}\ C\ \{q\}$, written $\vDash \{p\}\ C\ \{q\}$, if for all maximal traces $t \in \mathit{Traces}(C)$ with $\mathit{split}(t, \{o\}) = (ti, to)$ the following holds:*

$$\vDash_{FOL} p[\$/\mathit{sup}(ti)] \implies q[\$/\mathit{sup}(to)]$$

Example 11. The following specification of the Server class states that when a server is created and a request comes, the server creates a new worker and passes the worker the query and tells it to start propagating the result.

$\{\ \$ = \langle \underline{\text{this}} := \textbf{new}\ \texttt{Server} \cdot \underline{\text{this}}.\texttt{serve}(\underline{c}, \underline{y})\rangle\ \}$
Server
$\{\ \exists \underline{w} \bullet \$ = \langle \underline{w} := \textbf{new}\ \texttt{Worker} \cdot \underline{w}.\texttt{do}(\underline{y}) \cdot \underline{w}.\texttt{propagate}(\textbf{null}, \underline{c})\rangle\ \}$

From Def. 17 above, the semantics of the specification is a trace set of an object s of class Server, where for each maximal trace, the input and output parts are as stated in the specification. The trace set may include traces where all output events appear before the input events. In this case, the specification allows a worker to be created by the server before the server receives any request. However, the order of the output events must be as specified by the output trace assertion. This imprecision is as expected because the specification abstracts from the actual behavior of the implementation. In particular, the specification need not precisely describe the exact interleaving that happens.

When D is a component represented by its activator class $[C]$, the set of objects L of the run-time component needs to be extracted from the semantics. From Lemma 4, in any trace of $\mathit{Traces}([C])$ there is only one creation event of the initial object of the component. Thus, we can define the following function that extracts the set of objects of the run-time component that lies on the boundary of that run-time component.

Definition 18 (Boundary extraction). *Let T be a component trace set. $L' = \mathit{bound}(T)$ is the subset of objects of the run-time component, where*

$$bound(T) = \bigcup_{t \in T} bound(t),\ bound([\,]) = \emptyset,\ and$$

$$bound(t{\cdot}e) = \begin{cases} bound(t) \cup \{callee(e)\} & \text{if } msg(e) = \texttt{new}\ C \\ bound(t) \cup \{caller(e)\} \cup acq(e) - (acq(t) - bound(t)) & \\ \qquad \text{if } isMtd(msg(e)) \wedge callee(e) \notin bound(t) & \end{cases}$$

This function works similarly to the *idx* function (Def. 12), but it deals directly with a component trace set T as defined in Def. 15. The local object extraction of T is done by analyzing each trace t in T and combining the result of each analysis. If t ends with a creation event e, then the callee is part of the run-time component. By definition of the component trace semantics, there is exactly one creation event visible in any trace of T which is the creation of the initial object of the component instance. If t ends with a method call and it is directed to some foreign object, the caller of this event and all exposed objects in the method call arguments are included. Note that it is necessary to exclude foreign objects that were exposed to the component which is done by $acq(t) - bound(t)$. Using the boundary extractor function above, the semantics of $\{p\}\ [C]\ \{q\}$, called a *component triple*, is defined as follows.

Definition 19 (Component triple semantics). *Let* $[C]$ *represent a component and* $B = bound(Traces([C]))$ *the boundary objects of the component.* $Traces([C])$ *satisfies* $\{p\}\ [C]\ \{q\}$*, written* $\models \{p\}\ [C]\ \{q\}$*, if for all maximal traces* $t \in Traces([C])$ *with* $split(t, B) = (ti, to)$ *the following holds:*

$$\models_{FOL} p[\$/sup(ti)] \implies q[\$/sup(to)]$$

Example 12. As a component, the worker replies to the client by merging the partial result passed on to the component with the computation of the remaining subqueries as a whole. This property can be specified as follows.

$\{\ \$ = \langle \underline{\texttt{this}} := \texttt{new}\ \texttt{Worker} \cdot \underline{\texttt{this}}.\texttt{do}(\underline{\texttt{y}}) \cdot \underline{\texttt{this}}.\texttt{propagate}(\underline{\texttt{v}}, \underline{\texttt{c}}) \rangle\ \}$
$\quad [\texttt{Worker}]$
$\{\ \$ = \langle \underline{\texttt{c}}.\texttt{response}(\texttt{merge}(\underline{\texttt{v}}, \texttt{compute}(\underline{\texttt{y}}))) \rangle\ \}$

Similar to Ex. 11, the semantics of the specification above only deals with the maximal traces. The input trace consists of creating a new `Worker` object, obtaining the request to do a query and then propagating the computation result. When the input part is satisfied, the worker component produces a response back to the client by computing the whole remaining subqueries (following the assumption on `compute` given in Sect. 2.2) and merging it with the given partial result.

Usually, we want to cover as much as possible of the behavior of the entity we are specifying. Thus, its specification is a collection of triples. An implementation satisfies the specification, if its trace semantics satisfies all triples within the specification.

Definition 20 (Specifications). *Let* D *be a class or a component. A specification for* D *is a set of specification triples* $S = \{\{p_1\}\ D\ \{q_1\}, \ldots, \{p_n\}\ D\ \{q_n\}\}$*.* $Traces(D)$ *satisfies* S*, written* $\models S$*, if* $\forall(\{p_i\}\ D\ \{q_i\}) \in S \bullet \models \{p_i\}\ D\ \{q_i\}$*.*

Example 13. The worker class is described using two specification triples, each handling the base and inductive cases, respectively. The first triple handles the case when

$$\textsc{Consequence}\ \frac{p \implies p_1 \quad \{p_1\}\ D\ \{q_1\} \quad q_1 \implies q}{\{p\}\ D\ \{q\}} \qquad \textsc{Boxing}\ \frac{\{p\}\ C\ \{q \wedge nonCr(\$)\}}{\{p\}\ [C]\ \{q\}} \qquad \textsc{BoxedComposition}\ \frac{\{p \wedge \underline{i} = \$\}\ C\ \{q \wedge noSelfExp(\underline{i}, \$)\} \quad \{q'\}\ [C']\ \{r\} \quad match(q, q', C')}{\{p\}\ [C]\ \{r\}}\ \text{where } \underline{i} \notin free(p) \cup free(q)$$

$$\textsc{Induction}\ \frac{\{p \wedge m = 0\}\ [C]\ \{q\} \qquad match(p', p, C) \quad \{p \wedge m = \underline{z} \wedge m > 0 \wedge \underline{i} = \$\}\ C\ \{p' \wedge m < \underline{z} \wedge noSelfExp(\underline{i}, \$)\}}{\{p\}\ [C]\ \{q\}}\ \text{where } \underline{i} \notin free(p) \cup free(q)$$

$$\textsc{Invariance}\ \frac{\{p\}\ D\ \{q\}}{\{p \wedge r\}\ D\ \{q \wedge r\}}\ \text{where } consFree(r)$$

$$\textsc{Substitution}\ \frac{\{p\}\ D\ \{q\}}{\{p[\underline{x}/r]\}\ D\ \{q[\underline{x}/r]\}}\ \text{where } \underline{x} \in free(p) \cup free(q) \text{ and } consFree(r)$$

Fig. 7. Inference rules for PSA

the query has exactly one subquery. In this case, the worker sends back to the client the result of merging the propagated result value with the computation of the subquery.

$\{\ \$ = \langle \underline{this} := \textbf{new}\ \texttt{Worker} \cdot \underline{this}.\texttt{do}(\underline{y}) \cdot \underline{this}.\texttt{propagate}(\underline{v}, \underline{c}) \rangle \wedge \texttt{size}(\underline{y}) = 1\ \}$
$\quad \texttt{Worker}$
$\{\ \$ = \langle \underline{c}.\texttt{response}(\texttt{merge}(\underline{v}, \texttt{compute}(\underline{y}))) \rangle\ \}$

In the second case, the query consists of multiple subqueries. In this case, the worker creates another worker, passes on the rest of the subqueries, then processes the current subquery. When the computation of the current subquery is finished, the worker merges the computation result with the previous result it receives and propagates the merged result to the other worker.

$\{\ \$ = \langle \underline{this} := \textbf{new}\ \texttt{Worker} \cdot \underline{this}.\texttt{do}(\underline{y}) \cdot \underline{this}.\texttt{propagate}(\underline{v}, \underline{c}) \rangle \wedge \texttt{size}(\underline{y}) > 1\ \}$
$\quad \texttt{Worker}$
$\{\ \exists \underline{w} \bullet \$ = \langle \underline{w} := \textbf{new}\ \texttt{Worker} \cdot \underline{w}.\texttt{do}(\texttt{restQuery}(\underline{y})) \cdot$
$\qquad\qquad \underline{w}.\texttt{propagate}(\texttt{merge}(\underline{v}, \texttt{compute}(\texttt{fstQuery}(\underline{y}))), \underline{c}) \rangle\ \}$

5.2 Verification

The semantics of the specifications includes the open system aspect. Unlike the usual specification technique where one can refer to the program model, we have only the class or component name. To handle reasoning between class and component specifications, a special kind of proof system is needed. The proof system should allow one to verify component specifications by transitively inferring them from the class specifications. In this section, we provide a proof system that handles systems of similar nature to the running example to illustrate the complete picture of our approach.

The proof system PSA presented in Fig. 7 has a few inference rules. Each premise can be a trace assertion, which is applied to any trace using first-order logic semantics,

or a specification triple with the semantics as declared in Sect. 5.1. A *proof* is a tree of inference rule applications. Each node is a (possibly empty) premise and each edge is a rule application which in the following will be labeled with the applied inference rule. The root of the proof is the main goal: a specification triple. The leaves are either valid trace assertions or assumed class triples.

The rule CONSEQUENCE is a standard Hoare logic rule, where the input trace assertion can be weakened and the output trace assertion can be strengthened. This rule can be applied to a class or a component D.

BOXING transforms a class triple into a component triple, when the output trace assertion states that no object is created by the instance of that class. The output trace assertion uses the predicate *nonCr*, which checks for the lack of creation event in the output trace. This rule is derived from Def. 15 where the only object creation event that can be observed is that of the initial object of the component.

The BOXEDCOMPOSITION rule defines how an object o of class C can be combined with another component instance of initial class C' to create boxed component $[C]$. For this rule to be applicable, three premises must hold.

First, the class triple C must guarantee that the object identity will not be exposed.

$$noSelfExp(i, ec) \stackrel{\text{def}}{=} \forall \mathsf{e} \cdot ec' \in Pref(i) \bullet callee(\mathsf{e}) \notin acq(ec)$$

The predicate *noSelfExp*, short for *no self exposure*, takes variable i and the trace variable \$ representing the input and output traces, respectively. To ensure that no self exposure is made, the acquaintance of the output is checked against the callee of all events in the input trace, alias the object o. It guarantees one way interaction between o and objects of component $[C']$ because the current object is not exposed.

Second, the instance of C creates a component whose initial class is C'. Thus, no foreign object is created by the instance of $[C']$.

Third, the output produced by the object of class C must match the input of the instance of D. In other words, the object of class C exclusively feeds the instance of D in this particular case. This matching is handled by trace assertion *match*.

$$match(q, q', C') \stackrel{\text{def}}{=} q \implies \exists o \in \mathbf{O} \bullet firstCreated(o, \$) \wedge class(o) = C' \wedge q'$$

The predicate *firstCreated* checks if the first event is an object creation and o represents the created object. The predicate *classOf* checks if the created object is of class C'. The *match* assertion relies on the fact that a class or a component trace starts with an object creation (see Corollary 1, Lemma 4). This restriction applies because the evaluation of q' is done against an input trace, which always starts with an object creation. For *match* to hold, the free variables of q and q' should coincide. Because *match* is only used to link output trace assertion q to input trace assertion q', there is no need to explicitly check that q represents an output trace assertion. Recall that following the semantics of trace assertion *match* is checked against all traces, as it is used as a free standing trace assertion (i.e., not within a triple).

Example 14. Consider the output assertion of server class specification as described in Ex. 11 and the input assertion of the worker component as described in Ex. 12.

– $q = \exists \underline{w} \bullet \$ = \langle \underline{w} := \texttt{new Worker} \cdot \underline{w}.\texttt{do}(\underline{y}) \cdot \underline{w}.\texttt{propagate}(\texttt{null}, \underline{c}) \rangle$
– $q' = \$ = \langle \underline{\texttt{this}} := \texttt{new Worker} \cdot \underline{\texttt{this}}.\texttt{do}(\underline{y}) \cdot \underline{\texttt{this}}.\texttt{propagate}(\underline{v}, \underline{c}) \rangle$

If we instantiate $\underline{v}$ in q' with `null` (i.e., using the worker component to deal with a original request sent by the client), the output assertion q and input assertion q' match. Hence, $\vDash_{FOL} match(q, q'[\underline{v}/\texttt{null}], \texttt{Worker})$.

The last rule is INDUCTION. As the name suggests, this inference rule deals with the case when multiple objects of the same class are created to handle some input with parameters of method calls coming to those objects converging into a base case, as indicated by the measure variable m. It is similar to making a recursive call inside an object. The difference is that the concurrent nature of the objects causes each call may be processed independently by each created object, possibly optimizing the computation.

In addition to the inference rules above, PSA also includes some standard auxiliary rules: INVARIANCE and SUBSTITUTION. INVARIANCE allows a predicate containing no trace variable to strengthen both input and output trace assertions of a triple. SUBSTITUTION allows a free variable to be substituted to some predicate or expression r which must not contain the trace variable. As with CONSEQUENCE, these rules also apply for class and component triples.

Any meaningful proof system should be *sound*. A proof system is sound if and only if its inference rules only derive from some premises conclusions which are valid according to the given semantics. In our setting, sound means that each of the rules in PSA derives valid triples according to the corresponding trace semantics. Because of the open setting, proving the soundness of PSA is challenging. The soundness proof comes from our previous work.

Theorem 1 (Soundness [37]). *The proof system in Fig. 7 is sound.*

Using PSA, we can show that the server component replies back to a request from a client with the appropriate computation result. In the example below, we verify the server component triple from server class triple and worker component triple.

Example 15. A specification of the server component stating that a request from the client is replied by a response to the client with the computed result is as follows.

$\{ \$ = \langle \underline{\texttt{this}} := \texttt{new Server} \cdot \underline{\texttt{this}}.\texttt{serve}(\underline{c}, \underline{y}) \rangle \}$
$\quad [\texttt{Server}]$
$\{ \$ = \langle \underline{c}.\texttt{response}(\texttt{compute}(\underline{y})) \rangle \}$

Compared to the server class triple given in Ex. 11, there are two notable differences. The first is that the represented element is the component [Server]. Second, the output trace assertion directly deals with the expected output behavior of the component. Therefore, the specification hides how the server achieves the production of the output. The input assertion remains the same.

By applying the inference rules on the worker component (Ex. 12) and server class specifications (Ex. 11), we can infer the specification of the server component. That is, when the server is implemented in a way described by the worker component and the server class specifications, no matter how the server's context looks like, the server behaves as specified. We abbreviate the following assertions used in the specifications.

- $\mathsf{InSrv} = \mathsf{InSrvC} \stackrel{\text{def}}{=} \$ = \langle \underline{\mathsf{this}} := \textbf{new}\ \mathtt{Server} \cdot \underline{\mathsf{this}}.\mathtt{serve}(\underline{\mathsf{c}}, \underline{\mathsf{y}}) \rangle$
- $\mathsf{OutSrv} \stackrel{\text{def}}{=} \$ = \langle \underline{\mathsf{w}} := \textbf{new}\ \mathtt{Worker} \cdot \underline{\mathsf{w}}.\mathtt{do}(\underline{\mathsf{y}}) \cdot \underline{\mathsf{w}}.\mathtt{propagate}(\textbf{null}, \underline{\mathsf{c}}) \rangle$
- $\mathsf{InWrkC} \stackrel{\text{def}}{=} \$ = \langle \underline{\mathsf{this}} := \textbf{new}\ \mathtt{Worker} \cdot \underline{\mathsf{this}}.\mathtt{do}(\underline{\mathsf{y}}) \cdot \underline{\mathsf{this}}.\mathtt{propagate}(\underline{\mathsf{v}}, \underline{\mathsf{c}}) \rangle$
- $\mathsf{OutWrkC} \stackrel{\text{def}}{=} \$ = \langle \underline{\mathsf{c}}.\mathtt{response}(\mathtt{merge}(\underline{\mathsf{v}}, \mathtt{compute}(\underline{\mathsf{y}}))) \rangle$
- $\mathsf{OutSrvC} \stackrel{\text{def}}{=} \$ = \langle \underline{\mathsf{c}}.\mathtt{response}(\mathtt{compute}(\underline{\mathsf{y}})) \rangle$

The abbreviations are chosen such that InSrv, for example, represents the input event core equality of the server class triple, whereas OutWrkC represents the output event core equality of the worker class triple. The C suffix indicates the assertion is used in a component triple. We also introduce the function *cse*, short for *c*ore *s*equence *e*xtractor, to extract the event core sequences from these abbreviations.

To achieve the inference of the server component specification, we work backwards until the server class and worker component specifications are obtained. The suitable rule for this inference is BOXEDCOMPOSITION. The input to the server component is handled fully by the server object, while the output of the server object is captured completely by the worker component instance. To match the output trace assertion of the server class triple, the partial result variable in the input trace assertion of the worker component instance has to be initialized to **null**.

$$\textsc{Cmp}\ \frac{\begin{array}{c}\{\mathsf{InSrvC} \wedge \underline{\mathsf{i}} = \$\}\ \mathtt{Server}\ \{\exists \underline{\mathsf{w}} \bullet \mathsf{OutSrv} \wedge \mathit{noSelfExp}(\underline{\mathsf{i}}, \$)\} \\ \{\mathsf{InWrkC}[\underline{\mathsf{v}}/\textbf{null}]\}\ [\mathtt{Worker}]\ \{\mathsf{OutSrvC}\} \\ \mathit{match}(\exists \underline{\mathsf{w}} \bullet \mathsf{OutSrv}, \mathsf{InWrkC}[\underline{\mathsf{v}}/\textbf{null}], \mathtt{Worker})\end{array}}{\{\mathsf{InSrvC}\}\ [\mathtt{Server}]\ \{\mathsf{OutSrvC}\}}$$

As both triples left as proof obligation in the proof tree above are not of the assumed form, they must be transformed. The proof tree below shows how the server class triple above can be obtained from the original specification. We need to store the input trace and transfer it to the output trace assertion of the triple, to determine whether a reference to the created server object is not exposed. In this example, the core sequence extractor function is used to get the suppressed input trace that we need. Note that the input trace assertions of the server class and server component triples are the same.

$$\textsc{Cns}\ \frac{\textsc{Inv}\ \dfrac{\{\mathsf{InSrv}\}\ \mathtt{Server}\ \{\exists \underline{\mathsf{w}} \bullet \mathsf{OutSrv}\}}{\{\mathsf{InSrv} \wedge \underline{\mathsf{i}} = \mathit{cse}(\mathsf{InSrv})\}\ \mathtt{Server}\ \{\exists \underline{\mathsf{w}} \bullet \mathsf{OutSrv} \wedge \underline{\mathsf{i}} = \mathit{cse}(\mathsf{InSrv})\}} \quad \begin{array}{c}\mathsf{InSrv} \wedge \underline{\mathsf{i}} = \$ \implies \mathsf{InSrv} \wedge \underline{\mathsf{i}} = \mathit{cse}(\mathsf{InSrv}) \\ \exists \underline{\mathsf{w}} \bullet \mathsf{OutSrv} \wedge \underline{\mathsf{i}} = \mathit{cse}(\mathsf{InSrv}) \implies \exists \underline{\mathsf{w}} \bullet \mathsf{OutSrv} \wedge \mathit{noSelfExp}(\underline{\mathsf{i}}, \$)\end{array}}{\{\mathsf{InSrv} \wedge \underline{\mathsf{i}} = \$\}\ \mathtt{Server}\ \{\exists \underline{\mathsf{w}} \bullet \mathsf{OutSrv} \wedge \mathit{noSelfExp}(\underline{\mathsf{i}}, \$)\}}$$

The transformation for the worker component triple is straight forward to obtain. All occurrences of variable $\underline{\mathsf{v}}$ are substituted by **null**, which transforms the worker component output trace assertion into the server component output trace assertion. As all parts of the proof tree are closed, the proof of the server component triple is completed.

$$\textsc{Cns}\ \dfrac{\textsc{Sub}\ \dfrac{\{\mathsf{InWrkC}\}\ [\texttt{Worker}]\ \{\mathsf{OutWrkC}\}}{\{\mathsf{InWrkC}[\underline{\mathsf{v}}/\texttt{null}]\}\ [\texttt{Worker}]\ \{\mathsf{OutWrkC}[\underline{\mathsf{v}}/\texttt{null}]\}} \qquad \mathsf{OutWrkC}[\underline{\mathsf{v}}/\texttt{null}] \implies \mathsf{OutSrvC}}{\{\mathsf{InWrkC}[\underline{\mathsf{v}}/\texttt{null}]\}\ [\texttt{Worker}]\ \{\mathsf{OutSrvC}\}}$$

5.3 Discussion and Related Work

The specification technique can handle any triple where the output trace is exclusively determined by the input trace, that is, it fits to a triple $\{\underline{\mathsf{i}} = \$\}\ D\ \{R(\underline{\mathsf{i}}, \$)\}$ where R forms a trace assertion based on the input and output traces. Particularly, the input trace assertion may depend on the exposure of some objects which appear in the output trace assertion. However, the specification technique is incomplete, as it does not provide any information on the partial order between each message appearing in the input and the output traces [62]. Stated differently, the technique is sufficient when interleavings between the input and output traces are not important.

Techniques for specifying and verifying concurrent behavior are the subject of many well-known papers. The book by de Roever et al. [49] provides a detailed overview. Here we only focus on those more related to our work.

Several works have focused on concurrent object or actor models. Specification Diagram [54] provides a detailed, graphical way to specify how an actor system behaves. Our specification technique could be encoded into Specification Diagram by utilizing their *choice* operator to go through different kinds of interleaving between input and output events. However, to check whether a component specification produces the same behavior as the composition of the specification of its subcomponents one has to perform a non-trivial interaction simulation on the level of the state-based operational semantics. By extending π-calculus ([42]), a *May testing* ([19]) characterization of Specification Diagram can be obtained [59].

Ahrendt and Dylla [5] and Din et al. [20,22,21] extended Soundarajan's work to deal with concurrent object systems. They considered only finite prefix-closed traces, justifying it by having only finite number of objects to consider in the verification process. The work Din et al. extended previous work [23] where object creation is not treated. In particular, Din et al. verified whether an implementation of a class satisfies its triples by transforming the implementation in a simpler sequential language, applying the transformational method proposed by Olderog and Apt [45]. The main difference to our approach is the notion of component that hides a group of objects into a single entity. It avoids starting from the class specifications of each object belonging to a component when verifying a property of the component.

Other state-based specification and verification techniques for concurrent object systems have also been developed (e.g., [18,17,24,52,11]). However, they rely strongly on having the actual implementations, bypassing the intermediate tier that we would like to have. Apart from [17], they also require the knowledge of the environment.

Misra and Chandy [43], Soundarajan [55] and Widom et al. [62] proposed proof methods handling network of concurrent processes using traces. Misra and Chandy gave

a specification technique where the behavior of processes is described by using invariants on the traces. For each trace, a specification states an invariant over the next event that happens afterwards. The semantics of the specification relies on the prefix-closed properties of the trace semantics. Our specification technique differs from theirs by distinguishing the treatment of input and output events. Soundarajan related invariants on process histories of similar style to Misra and Chandy's to the axiomatic semantics of a parallel programming language. Widom et al. discussed the necessity of having prefix-closed trace semantics and partial ordering between messages of different channels to reach a complete proof system. For this reason, Misra and Chandy's proof system (and also ours) is incomplete. The setting used in these papers deal only with closed systems of fixed finite processes (and channels) and, because of their generality, make no use of the guarantees and restrictions of the concurrent object model.

De Boer [10] presented a Hoare logic for concurrent processes that communicate by message passing through FIFO channels in the setting of Kahn's deterministic process networks ([36]). He described a similar two-tier architecture, where the assertions are based on local and global rules. The local rules deal with the local state of a process, whereas the global rules deal with the message passing and creation of new processes. However, they only work for closed systems.

Classical models CSP [30], CCS [41] and π-calculus [42] allow specifying interactions among processes. Logics proposed, e.g., in [38] and [57] allow reasoning on CSP and CCS, respectively, while π-calculus models are usually analyzed by means of bisimulation. But because of their minimalistic approach, they are too abstract for two-tier verification. A possibility to apply these models is on the upper level tier (i.e., verifying component/system specifications from class specifications). However, showing the connection between component/system specifications and class specifications requires a significant adaptation to the setting. For example, Sangiorgi and Walker devoted a chapter in the final part of their book [51] to show how to restrict π-calculus to simulate object-orientation.

6 Conclusion

We have seen in this chapter a formal system that covers a part of open concurrent object systems. An attempt is made to describe what it means to compose classes (and smaller components) into components and how to use the composition to verify functional properties of the composed entity in an open setting. The notion of class composition chosen in this chapter is closely related to popular component frameworks. While it is clear that components should have interfaces and hide behavior, connecting these concepts to verification is not clear. The approach given here suggests that to obtain a sound verification technique which makes use of components, one needs to be explicit about the class of properties and how they are specified.

The investigation on how to apply the component notion for this verification purpose is still ongoing. While the connection between the first layer of verification (i.e., from the implementation to the class specification) given in the introduction is dealt with by the latest research [5,20,22], the connection between the proposed specification techniques is not yet explored. Furthermore, the presented proof system is useful for one

way pipeline communicating components. To extend the proof system into other cases, other common patterns of communication between components need to be considered. Two particular extensions of interest are the sound rules to compose more than two components/classes and two-way communication between components. And closer to ABS, it is also of interest to investigate patterns involving futures.

Acknowledgment. We thank Yannick Welsch and anonymous reviewers for suggestions to improve the presentation.

References

1. Ábrahám-Mumm, E., de Boer, F.S., de Roever, W.-P., Steffen, M.: Verification for Java's reentrant multithreading concept. In: Nielsen, M., Engberg, U. (eds.) FOSSACS 2002. LNCS, vol. 2303, pp. 5–20. Springer, Heidelberg (2002)
2. Ábrahám, E., Grabe, I., Grüner, A., Steffen, M.: Behavioral interface description of an object-oriented language with futures and promises. J. Log. Algebr. Program. 78(7), 491–518 (2009)
3. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
4. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. J. Funct. Program. 7(1), 1–72 (1997)
5. Ahrendt, W., Dylla, M.: A system for compositional verification of asynchronous objects. Sci. Comput. Program. 77(12), 1289–1309 (2012)
6. Armstrong, J.: Erlang. Commun. ACM 53, 68–75 (2010)
7. Arts, T., Dam, M.: Verifying a distributed database lookup manager written in Erlang. In: Wing, J.M., Woodcock, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 682–700. Springer, Heidelberg (1999)
8. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
9. Baker Jr., H.G., Hewitt, C.: The incremental garbage collection of processes. SIGART Bull., 55–59 (August 1977)
10. de Boer, F.S.: A Hoare logic for dynamic networks of asynchronously communicating deterministic processes. Theor. Comput. Sci. 274(1-2), 3–41 (2002)
11. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
12. Broy, M., Stølen, K.: Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement. Springer, New York (2001)
13. Cardelli, L.: Class-based vs. object-based languages. PLDI Tutorial (1996)
14. Cenciarelli, P., Knapp, A., Reus, B., Wirsing, M.: An event-based structural operational semantics of multi-threaded Java. In: Alves-Foss, J. (ed.) Formal Syntax and Semantics of Java. LNCS, vol. 1523, pp. 157–200. Springer, Heidelberg (1999)
15. Clarke, D., Östlund, J., Sergey, I., Wrigstad, T.: Ownership types: A survey. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming. LNCS, vol. 7850, pp. 15–58. Springer, Heidelberg (2013)
16. Clinger, W.D.: Foundations of Actor Semantics. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA (1981)
17. Dam, M., Fredlund, L.-Å., Gurov, D.: Toward parametric verification of open distributed systems. In: de Roever, W.-P., Langmaack, H., Pnueli, A. (eds.) COMPOS 1997. LNCS, vol. 1536, pp. 150–185. Springer, Heidelberg (1998)
18. Darlington, J., Guo, Y.: Formalising actors in linear logic. In: OOIS, pp. 37–53 (1994)

19. De Nicola, R., Hennessy, M.: Testing equivalences for processes. Theor. Comput. Sci. 34, 83–133 (1984)
20. Din, C.C., Dovland, J., Johnsen, E.B., Owe, O.: Observable behavior of distributed systems: Component reasoning for concurrent objects. J. Log. Algebr. Program. 81(3), 227–256 (2012)
21. Din, C.C., Dovland, J., Owe, O.: An approach to compositional reasoning about concurrent objects and futures. Research Report 415 (2012)
22. Din, C.C., Dovland, J., Owe, O.: Compositional reasoning about shared futures. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 94–108. Springer, Heidelberg (2012)
23. Dovland, J., Johnsen, E.B., Owe, O.: Verification of concurrent objects with asynchronous method calls. In: SwSTE, pp. 141–150. IEEE Computer Society (2005)
24. Duarte, C.H.C.: Proof-Theoretic foundations for the design of actor systems. Mathematical Structures in Computer Science 9(3), 227–252 (1999)
25. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
26. Gusfield, D.: Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology. Cambridge University Press (1997)
27. Hähnle, R.: The Abstract Behavioral Specification language: A tutorial introduction. In: de Boer, F., Bonsangue, M., Giachino, E., Hähnle, R. (eds.) FMCO 2012. LNCS, vol. 7866, pp. 1–37. Springer, Heidelberg (2013)
28. Haller, P.: On the integration of the actor model in mainstream technologies: The Scala perspective. In: AGERE! 2012, pp. 1–6. ACM, New York (2012)
29. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969)
30. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM 21(8), 666–677 (1978)
31. International Telecommunication Union: Open distributed processing – reference models parts 1–4. Tech. rep., ISO/IEC (1995)
32. Johnsen, E.B., Blanchette, J.C., Kyas, M., Owe, O.: Intra-Object versus Inter-Object: Concurrency and reasoning in Creol. Electr. Notes Theor. Comput. Sci. 243, 89–103 (2009)
33. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
34. Johnsen, E.B., Owe, O., Yu, I.C.: Creol: A type-safe object-oriented model for distributed concurrent systems. Theor. Comput. Sci. 365(1-2), 23–66 (2006)
35. Jonsson, B.: A fully abstract trace model for dataflow and asynchronous networks. Distributed Computing 7(4), 197–212 (1994)
36. Kahn, G.: The semantics of simple language for parallel programming. In: IFIP Congress, pp. 471–475 (1974)
37. Kurnia, I.W., Poetzsch-Heffter, A.: A relational trace logic for simple hierarchical actor-based component systems. In: AGERE! 2012, pp. 47–58. ACM, New York (2012), http://doi.acm.org/10.1145/2414639.2414647
38. Lamport, L., Schneider, F.B.: The "Hoare logic" of CSP, and all that. ACM Trans. Program. Lang. Syst. 6(2), 281–296 (1984), http://doi.acm.org/10.1145/2993.357247
39. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1996)
40. Microsoft: Component Object Model (COM) (January 1999), http://www.microsoft.com/com/default.asp
41. Milner, R.: A Calculus of Communicating Systems. Springer-Verlag New York, Inc., Secaucus (1982)

42. Milner, R.: Communicating and Mobile Systems – The π-Calculus. Cambridge University Press (1999)
43. Misra, J., Chandy, K.M.: Proofs of networks of processes. IEEE Trans. Software Eng. 7(4), 417–426 (1981)
44. Nain, S., Vardi, M.Y.: Trace semantics is fully abstract. In: LICS, pp. 59–68. IEEE Computer Society (2009)
45. Olderog, E.R., Apt, K.R.: Fairness in parallel programs: the transformational approach. ACM Trans. Program. Lang. Syst. 10(3), 420–455 (1988)
46. OMG: CORBA component model v4.0 (2006), http://www.omg.org/spec/CCM/
47. Philippsen, M.: A survey of concurrent object-oriented languages. Concurrency - Practice and Experience 12(10), 917–980 (2000)
48. Poetzsch-Heffter, A., Feller, C., Kurnia, I.W., Welsch, Y.: Model-based compatibility checking of system modifications. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012, Part I. LNCS, vol. 7609, pp. 97–111. Springer, Heidelberg (2012)
49. de Roever, W.P., de Boer, F.S., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Cambridge Tracts in Theoretical Computer Science, vol. 54. Cambridge University Press (2001)
50. Roth, A.: Specification and Verification of Object-Oriented Software Components. Ph.D. thesis, University of Karlsruhe (2006)
51. Sangiorgi, D., Walker, D.: The Pi-Calculus – A Theory of Mobile Processes. Cambridge University Press (2001)
52. Schacht, S.: Formal reasoning about actor programs using temporal logic. In: Agha, G., De Cindio, F., Rozenberg, G. (eds.) APN 2001. LNCS, vol. 2001, pp. 445–460. Springer, Heidelberg (2001)
53. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing active objects to concurrent components. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 275–299. Springer, Heidelberg (2010)
54. Smith, S.F., Talcott, C.L.: Specification diagrams for actor systems. Higher-Order and Symbolic Computation 15(4), 301–348 (2002)
55. Soundarajan, N.: A proof technique for parallel programs. Theoretical Computer Science 31(1-2), 13–29 (1984)
56. Steffen, M.: Object-Connectivity and Observability for Class-Based, Object-Oriented Languages. Habilitation thesis, Technische Faktultät der Christian-Albrechts-Universität zu Kiel, 281 pages (Jul 2006)
57. Stirling, C.: An introduction to modal and temporal logics for CCS. In: Yonezawa, A., Ito, T. (eds.) UK/Japan WS 1989. LNCS, vol. 491, pp. 1–20. Springer, Heidelberg (1991)
58. Talcott, C.L.: Composable semantic models for actor theories. Higher-Order and Symbolic Computation 11(3), 281–343 (1998)
59. Thati, P., Talcott, C., Agha, G.: Techniques for executing and reasoning about specification diagrams. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 521–536. Springer, Heidelberg (2004)
60. The OSGi Alliance: OSGi core release 5 (2012), http://www.osgi.org
61. Vasconcelos, V.T., Tokoro, M.: Traces semantics for actor systems. In: Zatarain-Cabada, R., Wang, J. (eds.) ECOOP-WS 1991. LNCS, vol. 612, pp. 141–162. Springer, Heidelberg (1992)
62. Widom, J., Gries, D., Schneider, F.B.: Completeness and incompleteness of trace-based network proof systems. In: POPL, pp. 27–38 (1987)
63. Yonezawa, A., Briot, J.P., Shibayama, E.: Object-oriented concurrent programming in ABCL/1. In: OOPSLA, pp. 258–268 (1986)

Automatic Inference of Bounds on Resource Consumption

Elvira Albert[1], Diego Esteban Alonso-Blas[1], Puri Arenas[1], Jesús Correas[1], Antonio Flores-Montoya[2], Samir Genaim[1], Miguel Gómez-Zamalloa[1], Abu Naser Masud[3], German Puebla[3], José Miguel Rojas[3], Guillermo Román-Díez[3], and Damiano Zanardini[3]

[1] Complutense University of Madrid (UCM), Spain
[2] Technische Universität Darmstadt (TUD), Germany
[3] Technical University of Madrid (UPM), Spain

Abstract. In this tutorial paper, we overview the techniques that underlie the automatic inference of resource consumption bounds. We first explain the basic techniques on a Java-like sequential language. Then, we describe the extensions that are required to apply our method on concurrent ABS programs. Finally, we discuss some advanced issues in resource analysis, including the inference of non-cumulative resources and the treatment of shared mutable data.

1 Introduction

Having information about the execution cost of programs, i.e., the amount of resources that the execution will require, is useful for many different purposes, including program optimization, verification and certification. Reasoning about execution cost is difficult and error-prone. Therefore, it is widely recognized that *cost analysis*, sometimes also referred to as *resource analysis* or *automatic complexity analysis*, is quite important. COSTA [46,45][1] is a state-of-the-art cost and termination analyzer which automates this task. The system is able to infer upper and lower bounds on the resource consumption of a large class of programs. Given a program P, the analysis results allow bounding the cost of executing P on any input data $\overline{x}$ without having to actually *run* $P(\overline{x})$.

The first successful proposal for *automatically* computing the complexity of programs was the seminal work of Wegbreit [42]. Since then, a number of cost analysis frameworks have been proposed, mostly in the context of *declarative* programming languages (functional programming [31,36,41,37,18] and logic programming [21,33]). Cost analysis of imperative programming languages has received significantly less attention. It is worth mentioning the pioneering work of [1]. To the best of our knowledge, COSTA has been the first system which automatically infers bounds on cost for a large class of Java-like programs, getting meaningful results. The system is implemented in Prolog (it runs both on

[1] Further information of the system is available at `http://costa.ls.fi.upm.es/~costa/costa/costa.php`

E. Giachino et al. (Eds.): FMCO 2012, LNCS 7866, pp. 119–144, 2013.

Ciao [26] and SWI Prolog [43]) and uses the Parma Polyhedra Library [17] for manipulating linear constraints.

1.1 Organization of the Tutorial

We use the classical approach to static cost analysis which consists of two phases. First, given a program and a *cost model*, the analysis produces *cost relations*, i.e., a system of recursive equations which capture the cost of the program in terms of the size of its input data. Section 2 overviews this first phase which requires, among other things, the translation of the imperative program into an intermediate representation and the inference of size relations. In a second phase the cost relations are solved into a closed-form, i.e., an expression which is not in recursive form and that can be directly evaluated. Section 3.1 describes our approach to infer closed-form upper bounds on the worst-case cost and Section 3.2 the techniques to infer closed-form lower bounds on the best-case cost. Known limitations of this classical approach are described in Section 3.3, where we also compare our approach with amortized cost analysis.

Section 4 overviews the extensions needed to infer the resource consumption of ABS programs [6] which adopt the concurrent objects concurrency model. The main challenge is in handling concurrent interleavings in a sound and precise way. This requires redefining the size analysis component for the concurrent objects model. Also, the fact that concurrent objects represent distributed components brings in a new notion of cost which is not monolithic (like in traditional sequential applications) but rather it captures the cost attributed to each distributed component separately. These two issues are explained in Section 4.1. The precision of the resource analysis of concurrent languages can be improved if we infer may-happen-in-parallel (MHP) relations that over-approximate the set of program points that may execute in parallel. Section 4.2 describes the MHP analysis integrated in COSTABS [4]. COSTABS is the extension of the COSTA system to analyze ABS programs.

Section 5 discusses advanced issues in resource analysis. We start by describing the analysis of memory consumption in Section 5.1. Memory consumption is different from other type of (cumulative) resources if the language has a garbage collector. We will see that the information on which objects are garbage collected can be integrated in the analysis. As a follow-up, we will discuss in Section 5.2 the inference of the task-level of a concurrent program which tries to over-approximate the number of tasks that can be simultaneously executing in a concurrent system. The similarity with the heap consumption analysis is that both types of resources are non-cumulative. Another advanced issue that we describe in Section 5.3 is the treatment of the shared mutable data in resource analysis. This is currently one of the main challenges in static analysis of object-oriented programs. Finally, Section 5.4 overviews the design of an incremental resource analysis which, given some previous analysis results and a change in a program, is able to recompute the analysis information by reanalyzing only the components affected by the changes.

Section 6 concludes and points out directions for future work.

2 From Programs to Cost Relations

This section describes how a program is analyzed in order to produce a *cost relation system* which defines its resource consumption. The analysis consists of a number of steps: (1) the program is transformed into a *rule-based representation* which facilitates the subsequent steps of the analysis without losing information about the resource consumption; (2) size analysis and abstract compilation are used to generate size relations which describe how the size of data changes during program execution; (3) the chosen cost model is applied to each instruction in order to obtain an expression which represents its cost; (4) finally, a cost relation system is obtained by joining the information gathered in the previous steps. Let us illustrate these steps by means of an example.

Rule-Based Intermediate Representation. The input language of the programs that COSTA analyze is Java bytecode [32]. The bytecode program is first transformed into a recursive intermediate representation (RBR) [8]. The transformation starts by constructing the Control Flow Graph (CFG) of the program. Each block of the CFG is transformed into one rule in the RBR. Iteration (i.e. for/while/do loops) is transformed into recursion and conditional constructs are represented as multiple (mutually exclusive) guarded rules. Bytecode instructions for method calls are transformed into the call of the corresponding rule in RBR and recursive method calls are thus transformed into recursions. These transformations determines the recursive structure of the resulting cost relation system (CR). Each rule in the RBR program will result in an equation in the CR. Intermediate programs resemble declarative programs due to their rule-based form. However, they are still imperative, since they use destructive assignment and store data in mutable data structures (stored in a global memory or heap).

Example 1. Fig. 1 shows at the top the Java source code of our running example. The Java code is shown only for clarity, since the analysis works directly on the bytecode. The example implements a sorting algorithm over an input array of integers. At the bottom of Fig. 1, the CFG and the RBR corresponding to the inner loop in the example are shown. The parameters in the rules of the RBR are tupled into input parameters corresponding to the variables on which they operate on, and the single output parameter corresponding to the return value of the rules. The CFG contains the bytecode instructions of the original input program. The entry rule to the loop is $while_1$. Its input arguments are the array `a` and local variables `j` and `v` and its output argument is the possibly modified array. Procedures $while_2$ and $while_3$ correspond to the two conditions of the loop and both are defined by two mutually exclusive guarded rules. The iterative structure of the loop is preserved by the recursive call to $while_1$ in the second $while_3$ rule.

Cost Models. A *cost model* $\mathcal{M}$ determines the cost (a natural number) of each basic instruction b of the language. COSTA incorporates, among others, the following cost models:

```
static void sort(int a[]) {
  for (int i = a.length-2; i>=0; i--)
    {
 (q)  int j = i+1;
      int v = a[i];
      while ( j<a.length && a[j]<v ) {
 (p)    a[j-1] = a[j];
        j++;
      }
      a[j-1]=v;
  }
}
```

$$
\begin{aligned}
&while_1(\langle a,j,v\rangle,\langle a'\rangle) \leftarrow while_2(\langle a,j,v\rangle,\langle a'\rangle).\\
&while_2(\langle a,j,v\rangle,\langle a\rangle) \leftarrow j{\geq}a.length.\\
&while_2(\langle a,j,v\rangle,\langle a'\rangle) \leftarrow j{<}a.length,\\
&\qquad\qquad while_3(\langle a,j,v\rangle,\langle a'\rangle).\\
&while_3(\langle a,j,v\rangle,\langle a\rangle) \leftarrow a[j]{\geq}v.\\
&while_3(\langle a,j,v\rangle,\langle a'\rangle) \leftarrow a[j]{<}v,\\
&\qquad\qquad a[j-1]{=}a[j], j{=}j+1,\\
&\qquad\qquad while_1(\langle a,j,v\rangle,\langle a'\rangle).
\end{aligned}
$$

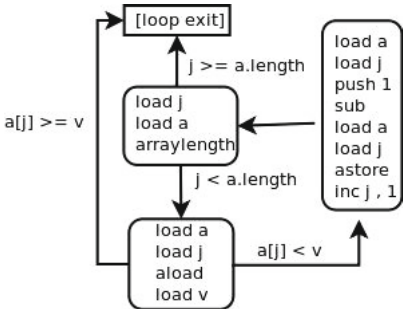


Fig. 1. Running Java example, Control Flow Graph and RBR

- *Number of instructions*: we have that $\mathcal{M}_i(b) = 1$, i.e., each bytecode instruction b in the rule-based program counts 1;
- *Number of calls to a method*: calls to methods in bytecode are of the form `invoke method_name`, thus $\mathcal{M}_c(b) = 1$ if $b \equiv$ `invoke m`; otherwise $\mathcal{M}_c(b) = 0$.
- *Heap consumption*: $\mathcal{M}_h(b) = size(\texttt{C})$ if $b \equiv$ `new C`, otherwise $\mathcal{M}_c(b) = 0$, where $size(\texttt{C})$ returns the amount of memory allocated in the heap when executing `new C`.

Generation of Cost Relations. Given a program P (in RBR form) and a cost model $\mathcal{M}$, we automatically generate a *cost relations system* (CR) which defines the cost of executing the program on some input $\overline{x}$ w.r.t. the selected cost model $\mathcal{M}$. CR are basically an extended form of recurrence relations. A CR is defined by a finite set of equations of the form $\langle c(\bar{x}) = e, \varphi\rangle$, where e is a cost expression and φ is a set of linear constraints which define the applicability conditions for the equations and the size relations among the variables. Variables in the equations represent the "sizes" of the corresponding data in the program according to the selected *size measure* [19]. Each program variable is abstracted using a size measure such that every non-integer value is represented as a natural number. Classical size measures used for non-integer types are: *array length* for arrays, the length of the longest reference path for linked data structures, etc. In our running example, arrays are abstracted to their length. Thus, variable a in the

CR represents the length of the array a. Note that due to this choice of size abstraction, we cannot observe the values stored in the elements of the array. All information about them is not present in the CR.

The inference of φ is defined as a *fixpoint computation* which comprises two steps: First, *abstract compilation* of the program replaces bytecode instructions in the RBR by size constraints. Then, *size analysis* infers *size-relations* between the states at different program points, i.e., it approximates how the sizes of variables change from one call in the cost relation to another. Analysis is often done by obtaining an abstract version of the program by relying on abstract interpretation [20]. In Fig. 2, such size relations are shown to the right of the equations.

Example 2. Let us assume that ⓠ (resp. ⓟ) represents the cost of executing the body of the `for` excluding the cost of the inner loop (resp. the cost of executing the body of the `while` loop). Here, ⓠ and ⓟ are symbolic cost expressions (or constants) and we assume this for simplicity at this stage. Fig. 2 shows the resulting CR for the running example of Fig 1. Here, variables are constraint variables corresponding to those of the original rule, e.g. i and i' both correspond to values of variable `i`, but at different program points. Instructions are replaced by linear constraints. Array `a` is abstracted to its length a. Thus, the first rules for $while_1$ become non-deterministic, as it is not possible to observe the array elements. Output variables are removed by inferring input-output size relations.

$$
\begin{array}{rll}
sort(a) = for(a,i) & & \{i{=}a{-}2,\ a{\geq}0\} \\
for(a,i) = 0 & & \{i{<}0\} \\
for(a,i) = \text{ⓠ}{+}while_1(a,j){+}for(a',i') & & \{i{\geq}0, j{=}i{+}1, i'{=}i{-}1, a'{=}a\} \\
 & & \\
while_1(a,j) = 0 & & \{j{\geq}a\} \\
while_1(a,j) = 0 & & \{j{<}a\} \\
while_1(a,j) = \text{ⓟ}{+}while_1(a',j') & & \{j{<}a, j'{=}j{+}1, a'{=}a\}
\end{array}
$$

Fig. 2. CR from example in Fig 1

In practice, the process of generating cost relations additionally involves several other static analysis techniques. In particular, *class analysis* is performed to compute the reachable code; *nullity* and *array bound* analysis for dead code elimination, *slicing* to eliminate variables which are irrelevant to cost and *cyclicity* analysis to identify cyclic data structures.

3 From Cost Relations to Closed-Form Bounds

Though CRs are simpler than the programs they originate from, since all variables are of integer type, in several respects they are not as static as one would

expect, namely: (1) in order to obtain a concrete bound, for a given input, we still need to evaluate the CRs, and moreover, due to the nondeterministic nature of CRs, such evaluation must consider all possibilities and select the worst/best among them; and (2) CRs do not provide a clear insight on the complexity class to which the resource consumption belongs, e.g., polynomial, exponential, etc. For these reasons, *closed-form* bounds, which are functions composed from simple arithmetic expressions, are preferable. This form makes it possible to address the above two issues efficiently. COSTA includes a solver called PUBS, that is encharged of solving CRs into closed-form upper and lower bounds.

Solving CRs into closed-form (lower or upper) bounds in PUBS is done in two phases. In the first phase, the CRs are simplified such that all recursions are direct (i.e. cycles in the call graph are of length one), which is achieved by applying *partial evaluation* [29] in order to unfold intermediate equations. After this step, each iterative or recursive construct in the original program is represented by a single directly recursive CR. The CRs of Fig. 2 are given in this form. In the second phase, the CRs are solved into closed-form bounds in a compositional way, by handling one CR at a time, in a reversed order to the calling relation. E.g., for the CRs of Fig. 2 the solving proceeds as follows: it solves CR $while_1$; substitutes the resulting bound in the CR *for*; solves CR *for*; substitutes the resulting bound in CR *sort*; and finally solves CR *sort*. These two phases are common for inferring both upper and lower bounds, the difference is in how each CR (that does not call any other CRs, i.e., standalone), in the second phase above, is solved. This is discussed in sections 3.1 and 3.2. Sec. 3.3 discusses programs whose cost cannot be modeled precisely with CRs, and explain a corresponding solution.

Note that a common feature to all solving methods, that we describe in this section, is that they heavily rely on the use of program analysis techniques. This, we believe, is the most important factor that made COSTA succeed where other previous cost analyses had failed.

3.1 Upper Bounds

PUBS includes two approaches for solving CRs into closed-form upper-bounds. They have different applicability and precision properties. In what follows we explain the essentials of both approaches.

We start by intuitively explaining the first approach using the CRs of Fig. 2, starting with the standalone CR $while_1$. Let a_0 and j_0 be unknown initial values, for a and j respectively, with which $while_1$ is called. Solving this CR is done by inferring an upper-bound $\hat{f}(a_0, j_0)$ on the number of times that the recursive equation can be applied, when evaluating $while_1(a_0, j_0)$, then, $\hat{f}(a_0, j_0) * \textcircled{p}$ is guaranteed to be an upper-bound for $while_1(a_0, j_0)$ since all applications of the recursive equation contribute the constant symbol $\textcircled{p}$. Inferring $\hat{f}(a_0, j_0)$ automatically can be done by relying on techniques from the field of termination analysis, such as *synthesis of ranking functions* [34]. For the case of $while_1$, we automatically infer $\hat{f}(a_0, j_0) = \mathtt{nat}(a_0 - j_0)$, where $\mathtt{nat}(v) = \max(v, 0)$, and thus,

$\texttt{nat}(a_0 - j_0) * \textcircled{p}$ is an upper-bound for $while_1(a_0, j_0)$. The `nat` function is used to lift negative values to zero because negative evaluations of $a_0 - j_0$ means that the number of times the recursive equation of $while_1$ applied in an evaluation is zero.

Next we proceed to solve the CR *for*. First we substitute the upper-bound of $while_1$ in the CR of *for*, which converts it into standalone

$$\begin{array}{ll} for(a,i) = 0 & \{i{<}0\} \\ for(a,i) = \textcircled{q} + \texttt{nat}(a-j) * \textcircled{p} + for(a',i') & \{i{\geq}0, j{=}i{+}1, i'{=}i{-}1, a'{=}a\} \end{array}$$

In this case, $\hat{f}(a_0, i_0) = \texttt{nat}(i_0 - 1)$ is an upper-bound on the number of times that the recursive equation can be applied, when evaluating $for(a_0, i_0)$. Assuming that we have a function $\hat{e}(a_0, i_0)$ such that it is guaranteed to be bigger than any instance of $\textcircled{q} + \texttt{nat}(a - j) * \textcircled{p}$, then $\hat{f}(a_0, i_0) * \hat{e}(a_0, i_0)$ is an upper-bound for $for(a_0, i_0)$. Let us explain how to automatically compute $\hat{e}(a_0, i_0)$. Since $\textcircled{q} + \texttt{nat}(a - j) * \textcircled{p}$ takes its maximum values when $a - j$ is maximal, it is enough to compute an upper-bound on $a - j$ (in terms of a_0 and j_0). This can be done using invariant generation and linear programming as follows: (1) we infer an invariant Ψ that relates the initial values a_0 and i_0 to the values of a and i at any call $for(a, i)$, which is $\Psi = \{a{=}a_0, i_0{\geq}i\}$ in this case; and (2) the maximum value to which $a - j$ can be evaluated is obtained applying the recursive equation in the context of Ψ, and asking what is the maximum of $a - j$ for this application. This is equivalent to solving the following parametric integer programming [22] problem:

$$\texttt{maximize}\ a{-}j\ \texttt{w.r.t}$$
$$\{a{=}a_0, i_0{\geq}i\} \wedge \{i{\geq}0, j{=}i{+}1, i'{=}i{-}1, a'{=}a\}\ \text{and the}\ \texttt{parameters}\ a_0, i_0$$

which results in $a_0 - 1$. Then, $\hat{e}(a_0, i_0) = \textcircled{q} + \texttt{nat}(a_0 - 1) * \textcircled{p}$, and thus $\texttt{nat}(i_0 + 1) * (\textcircled{q} + \texttt{nat}(a_0 - 1) * \textcircled{p})$ is an upper-bound for CR *for*. An upper-bound $\texttt{nat}(a_0 - 1) * (\textcircled{q} + \texttt{nat}(a_0 - 1) * \textcircled{p})$ for CR *sort* is then obtained by substituting the one of *for* in the corresponding equation. Note that this approach is general enough to handle the CRs which is constructed with possible multiple equations having possible multiple recursive calls. E.g, if CR $while_1$ had two recursive calls, then it would obtain the upper-bound $2^{\texttt{nat}(a_0 - j_0)} * \textcircled{p}$ – For more details see [6]. Note also that PUBS provides a mechanism for converting the above non-asymptotic bounds to asymptotic ones [2], e.g., for the case of *sort* it computes $O(\texttt{nat}(a_0)^2 * \textcircled{p} + \texttt{nat}(a_0) * \textcircled{q})$.

In the above approach, the contributions of all applications of the recursive equation are approximated by the same amount. E.g., in the case of CR *for*, all instances of $\textcircled{q} + \texttt{nat}(a - j) * \textcircled{p}$ are approximated by $\textcircled{q} + \texttt{nat}(a_0 - 1) * \textcircled{p}$, while in practice this happens only in the last application, when $j = 1$ (i.e., when $i = 0$ since $j = i + 1$). This leads to some imprecision that might be crucial for some applications. To overcome this imprecision, PUBS provides an alternative approach that is based on simulating the contributions of the different applications using sequences (arithmetic, geometric, etc.). E.g., in the case of CR

$$
\begin{array}{ll}
\mathit{while}_1(a,j,\lambda) = 0 & \{j \geq a\} \\
\mathit{while}_1(a,j,\lambda) = 0 & \{j < a\} \\
\mathit{while}_1(a,j,\lambda) = \textcircled{p} + \mathit{while}_1(a',j',\lambda') & \{j < a, j' = j+1, a' = a, \lambda' = \lambda + 1\}
\end{array}
$$

Fig. 3. CR after instrumenting counter variables in the CRs in Fig 2

for, it first infers that the difference between the contributions of two consecutive application is at least $\hat{d} = 1$, and then it considers the arithmetic sequence $\hat{u}_1 = \hat{e}(a_0, i_0)$, $\hat{u}_i = \hat{u}_{i-1} - \hat{d}$, and sum the first $\hat{f}(a_0, i_0)$ elements (which are positive) of this sequence. This sum is guaranteed to be an upper-bound for $\mathit{for}(a_0, i_0)$ since the equation is applied at most $\hat{f}(a_0, i_0)$ times, and moreover the sequence starts from the maximum value $\hat{e}(a_0, i_0)$. For the case of CR *for* we obtain $\frac{1}{2} * \textcircled{p} * (\mathtt{nat}(a_0 - 1) * (\mathtt{nat}(a_0 - 1) - 1)) + \textcircled{q} * \mathtt{nat}(a_0 - 1)$ which is more precise than what we have obtained above. Note, however, that they are asymptotically equivalent. In practice, the summation is computed by solving a corresponding recurrence relation using a computer algebra system – For more details see [14].

3.2 Lower Bounds

In the latter approach [14], the inference of LBs is a dual problem to that of inferring UBs. The main difference is that one has to use (new) techniques for inferring LBs on the number of iterations and obtaining the best-case cost of each iteration. LB on the number of iterations can be inferred by instrumenting the given CR and inferring an invariant on that CR. First, the arguments of each head of the given CR are augmented with a new counter variable λ that is incremented by 1 in each recursive call of that CR. Next, an invariant Ψ is inferred for this new CR such that Ψ holds between an initial call with 0 as the initial counter value and any other call to the new CR with counter variable λ. Then, the LB on the number of iterations is obtained by minimizing λ w.r.t Ψ and φ_0, where φ_0 is the set of constraints in the base-case of the new CR. Minimization of λ can be done using parametric integer programming or by looking syntactically $\lambda \geq l$ in $\Psi \wedge \varphi_0$ where l is over the initial arguments.

Example 3. Let us consider the CR while_1 in Fig. 2. We now instrument it with the counter variable λ as shown in Fig. 3. The invariant Ψ between an initial call $\mathit{while}_1(a_0, j_0, 0)$ and another call $\mathit{while}_1(a, j, \lambda)$ is $\Psi \equiv \{j \geq j_0, a = a_0, \lambda = j - j_0\}$. λ is minimized to 0 w.r.t Ψ and the base-case constraints $(j \geq a \vee j < a)$. The LB on the iterations of *for* obtained is $nat(i_0 + 1)$ where i_0 is the initial value of i.

The best-case cost in each iteration is obtained by transforming the given CR into a best-case recurrence relation (RR for sort). Suppose $\langle C(\bar{x}) = e + C(\bar{x}'), \varphi \rangle$ be any CR and $e_1, \cdots, e_n$ be the costs contributed by e along the n iterations of $C(\bar{x})$. In order to obtain the LB cost for C, first, a LB $\check{n}$ on n (i.e. $\check{n} \leq n$)

is obtained (as above). Then, a series of costs $u_1, \cdots, u_{\check{n}}$ such that $u_i \leq e_i$ for all $1 \leq i \leq \check{n}$ are obtained, and then $u_1 + \cdots + u_{\check{n}}$ is the LB of C. When e is a simple linear expression, the novel idea is to view $u_1, \cdots, u_{\check{n}}$ as an arithmetic sequence that starts from $u_1 \equiv \check{e}$ and each time increases by $\check{d}$, i.e., $u_i = u_{i-1} + \check{d}$ where $\check{e}$ is the minimization of e, and $\check{d}$ is an under-approximation of all $d_i = e_{i+1} - e_i$. Minimization of e, i.e. $\check{e}$, can be obtained by using parametric integer programming w.r.t an appropriate invariant. When e is a complex non-linear expression, e.g., $l * l'$, it can be approximated by approximating each sub-expressions (which are linear) separately. Technically, the summation $u_1 + \cdots + u_{\check{n}}$ can be approximated by transforming the CR into a RR whose closed-form solution is the LB cost after instantiating the recurrence counter by $\check{n}$.

Example 4. Let us consider again the CR in Fig. 2. The LB cost of $while_1$ is 0 since the LB on the iterations of $while_1$ is 0 (see example 3). After substituting the cost of $while_1$, the recursive equation for CR *for* is $\langle for(a,i) = \textcircled{q} + for(a',i'), \{i \geq 0, j = i+1, i' = i-1, a' = a\}\rangle$. Here, $\check{d} = 0$ as cost $\textcircled{q}$ is constant and $\check{n} = nat(i_0 + 1)$. Next, the RR of *for* is $\langle P_{for}(0) = 0,\ P_{for}(N) = \textcircled{q} + P_{for}(N-1)\rangle$ whose closed-form solution is $E = \textcircled{q} * N$, and LB cost is $\textcircled{q} * nat(i_0 + 1)$ (after replacing N by $nat(i_0 + 1)$ in E). Finally, the LB cost of $sort(a_0)$ is $\textcircled{q} * nat(a_0 - 1)$. This is the LB that we have expected, since when the array is sorted, the inner loop does not perform any iteration and the best-case cost is linear on the length of the array.

3.3 Amortised Cost Analysis

The classical approach of COSTA is based on assuming that the cost of a procedure is solely determined by the size of its *input* data. In some procedures, there is also a codependency between the outputs and the cost, which may be crucial to infer precise cost bounds. Yet, since CRS do not model this codependency, for such programs the COSTA approach necessarily infers imprecise bounds.

Example 5. Consider the program of Fig. 4 (left), adapted from [40], where $*$ in the guard of the *while* loop corresponds to a nondeterministic evaluation of *true* or *false*. This nondeterministic choice is reflected in the constraints of equation $rpop(s) = 0$ in Fig. 4 (right) as the while loop can terminate for any value of $s \geq 0$. The procedure `main` admits the UB $\textcircled{r} * s$, but COSTA gets the asymptotically imprecise UB $\textcircled{r} * s * m$ instead. The reason is that the nondeterministic procedure *rpop* sets up a codependency between its cost and s', its return value: a possible execution of `rpop(s)` consumes $\textcircled{r} * s$ and returns $s'=0$; another one returns $s'=s$ and consumes zero; but no one both consumes $\textcircled{r} * s$ and also returns $s'=s$. COSTA abstracts the program into the CRS at Fig. 4 (right up), solves $rpop(s)$ into the precise bound $s * \textcircled{r}$ and unfolds this bound in the *main* CRS (right down). Although both this UB and the postcondition $s \geq s' \geq 0$ are precise abstractions w.r.t. `rpop`, they miss the described output-cost codependency. Thus, the CRS semantics now includes the spurious case of an execution of `rpop` consuming $\textcircled{r} * s$ and returning $s' = s$. For this reason, the

```
int rpop(int s){
    while(s > 0 && * )
        s-- ;
    return s ;
}
void main(int s,int m){
    for(;m>0;m--)
        s = rpop(s);
}
```

$$
\begin{array}{rll}
rpop(s) = & 0 & \{s \geq 0\} \\
rpop(s) = & \textcircled{r} + rpop(s-1) & \{s \geq 1\} \\
main(s,m) = & 0 & \{m = 0, s \geq 0\} \\
main(s,m) = & rpop(s) + \\ & main(s', m-1) & \left\{ \begin{array}{l} m \geq 1 \\ s \geq s' \geq 0 \end{array} \right\}
\end{array}
$$

$$
\begin{array}{rll}
main(s,m) = & 0 & \{m = 0, s \geq 0\} \\
main(s,m) = & \textcircled{r} * s + \\ & main(s', m-1) & \left\{ \begin{array}{l} m \geq 1 \\ s \geq s' \geq 0 \end{array} \right\}
\end{array}
$$

Fig. 4. Example of Amortised cost, with the Java program (left), the inferred CRS (right up) and the CRS unfolding the UB for *rpop* (right down)

CRS does not admit the bound $\textcircled{r} * s$ that we look for, and the techniques of Section 3.1 give $\textcircled{r} * s * m$ as the most precise UB for the *main* CRS.

Examples like this usually appear in the context of *amortised cost analysis* [40]. There, the output-cost codependency is described like the variable s storing *credit* or *potential* to pay the decrement operations. To overcome these limitations, we have recently developed [16] a novel definition of UBs that involve input and output arguments: a net-cost UB $\tilde{rpop}(s|s')$ bounds the cost of any terminating evaluation of *rpop* from an input s to an output s'. By making s' an input of the UB, net-cost UBs capture the output-cost codependency. In [16] we also describe a solving procedure based on real quantifier elimination, and draw a relation between net-cost functions and the potential functions used in the *automated amortised approach* [27,30,35].

4 Concurrency and Distribution

In order to develop a resource analysis for distributed and concurrent programs, we have considered a concurrency model based on the notion of concurrently running (groups of) objects, similar to the actor-based and active-objects approaches [38,39]. These models take advantage of the concurrency implicit in the notion of object in order to provide programmers with high-level concurrency constructs that help in producing concurrent applications more modularly and in a less error-prone way. The main novelty of the analysis is that it provides the resource consumption per *cost center*, where each cost center represents a distributed component. Having anticipated knowledge on the resource consumption of the different components which constitute a system is useful for distributing the load of work. Upper bounds can be used to predict that one component may receive a large amount of remote requests, while other siblings are idle most of the time. Also, our framework allows instantiating the different components with the particular features of the infrastructure on which they are deployed.

```
def B look⟨A,B⟩(Map⟨A,B⟩ ms, A k) =
  case ms {Ass(Pair(k,y),_) ⇒ y;
           Ass(_,tm) ⇒ look(tm,k);
}
class AddrBook {
  Map⟨String,User⟩ users = EmptyM;
  User getUser(String email){
    return look(users,email);
  }
}
class User {
  List⟨String⟩ msgs = Nil;
  Unit receive(String m) {
    msgs = Cons(m,msgs);
  }
}
```

```
class MailServer(AddrBook ab) {
  List⟨String⟩ emails =Nil;
  Unit addUser(String email) {
    emails = Cons(email, emails);
  }
  Unit addUsers(List⟨String⟩ l) {
    while ( l != Nil ) {
      this ! addUser(head(l));
      l=tail(l);
    }
  }
  Unit notify(String m) {
    while (emails != Nil) {
      Fut⟨User⟩ u;
      u = ab ! getUser(head(emails));
      await u ? ;
      User us = u.get;
      us ! receive(m);
      emails = tail(emails);
    }
  }
}
```

Fig. 5. ABS Implementation of a Mail Server

4.1 The Basic Cost Analysis Framework for Concurrency

ABS [28] is an *a*bstract *b*ehavioral *s*pecification language for distributed object-oriented systems. COSTA has been recently extended to be able to infer meaningful bounds for ABS programs [3]. The main novelties are related to the concurrency and distribution aspects of the language. *Concurrency* poses new challenges to the process of obtaining sound and precise size relations. This is mainly because the interleaving behaviour inherent to concurrent computations can influence how the sizes of data are modified. *Distribution* does not match well with the traditional monolithic notion of cost which aggregates the cost of all distributed components together. We use *cost centers* to keep the resource consumption of the different distributed components separate. An implementation of this cost analysis framework is described in [4]. The system is open-source and can be downloaded (together with examples, documentation, etc.) from: `http://costa.ls.fi.upm.es/costabs`.

Example 6. The example in Fig. 5 shows a simple mail server application programmed in ABS. Due to lack of space, we omit data and type definitions. At the top, we see a fragment of the functional subprogram which includes the function *look*. The imperative concurrent part contains the implementation of all classes. Calls to functions and functional data structures appear in italics. A mail server is composed of an address book (the class parameter ab) and a list of email addresses (the field emails). Email addresses can be added to the server by

invoking method addUser and addUsers. Method notify sends a message (m) to all users in the list emails. To this end, it first asynchronously invokes getUser in order to retrieve the next user (variable u) in the list. The await instruction allows releasing the processor if the information is not ready. The next instruction get blocks the execution of the current task until the requested information has arrived. When it arrives, the asynchronous call to receive is encharged of sending the message to the corresponding user without any kind of synchronization.

In what follows, we explain briefly the differences w.r.t. cost analysis in a sequential setting on the running example:

Cost Models for Concurrency. We consider the cost models *steps*, *memory*, *objects* and *task-level*. The first two ones are inherited from the sequential setting (see Section 2), while the last two ones are specific for concurrency. The "objects" cost model counts the total number of objects created along the execution. This provides an indication of the amount of parallelism that might be achieved, since each object could be running in a different processor. The "task-level" cost model estimates the number of tasks that are spawned along an execution. This can be counted by tracing how many asynchronous calls are performed. The task-level is useful for finding optimal deployment configurations, and detect situations like when one component is receiving too many requests while its siblings are idle.

Size Analysis. In order to handle the concurrency primitives, the classical sequential size analysis described in Section 2 is modified as follows: (a) when executing an instruction which does not cause the suspension of the current task, then fields (i.e., the global state) are tracked as if they were local variables, since in the concurrent objects setting it is guaranteed that in such circumstances no other tasks can modify those fields simultaneously; and (b) when executing an instruction that might cause suspension (e.g., `await`) of the current task, then the analysis loses all information about the corresponding fields, this is because they might be modified by other tasks in the meantime. This simple modification guarantees soundness of size analysis for a concurrent setting. However, it often loses precision. For example, in the while loop of method notify, losing the information on the field emails when executing `await` prevents us from proving that its size decreases in each iteration. Thus, the technique fails to bound the number of iterations of that loop. To overcome this problem, we provide a way to incorporate class invariants. For example, if we add the following invariant (using JML syntax) `//@invariant \old(emails) == emails` before the `await` instruction in the while loop of method notify, then we state that it is guaranteed that when the process resumes, the value of emails will be the same as when the process has been suspended. At present, we can infer class invariants automatically in a limited manner (See [3] for details). For example, if a variable (or shared location) is initialized and is never updated afterwards, we can infer that the values of this variable before and after a release point are always equal.

Cost Centers. The last step in this framework uses the inferred size relations and the selected cost model in order to generate cost equations and solve them into closed-form bounds. See Section 2 for more details. Now, let us explain the upper bounds that we obtain for the running example.

By applying the analysis starting from method notify and using the *steps* cost model, we obtain the following upper bound (after simplifying the constants for the sake of readability): $5 + (22 + 4 * users^{+}) * emails^{+}$. Variables $emails^{+}$ and $users^{+}$ refer to the maximum sizes of the fields emails and users respectively. The subexpression $(22 + 4 * users^{+})$ refers to the cost of each iteration of the while loop. Note that the subexpression $4 * users^{+}$ refers to the cost consumed by function *look*. The constant 4 is for executing the code of *look* once, and $users^{+}$ is the number of recursive calls. The cost of each iteration is then multiplied by $emails^{+}$, which is a bound on the number of iterations of the while loop. Finally, we add 5 to account for the cost of the instructions outside the loop (in this case it refers to the last comparison of the while loop's guard).

Instead of computing a monolithic cost expression, there exists the option of splitting the cost into *Cost Centers* that represent the different distributed components of the system. By assuming that objects of the same type belong to the same cost center, we obtain the following upper bounds (after simplification of constants for the sake of readability):

Cost Center	Upper Bound
MailServer	$5 + 16 * emails^{+}$
User	$3 * emails^{+}$
AddrBook	$(3 + 4 * users^{+}) * emails^{+}$

Observe that the sum of these bounds is identical to the single bound we have obtained before.

4.2 MHP

In the previous section, an invariant was used to ensure that the list of emails was not modified during the await. However, in order for the analysis to be safe, this invariant must be proven correct. An invariant in an `await` instruction expresses properties of the object fields that are maintained during the execution of the `await`. Therefore, the validity of these invariants depends on the actual changes that can take place while the current task is suspended.

A first step towards verifying these invariants consists on approximating the instructions that can be executed at that point. In our example, the invariant `//@invariant \old(emails) == emails` will hold if the instructions that can be executed during the `await` do not modify the field *emails*. For that purpose, a may-happen-in-parallel analysis has been developed [11,12].

To illustrate the behavior of the MHP analysis, we complete our example code with the main block in Figure 6 that defines the entry point of the program. The main block implements the following usage scenario: (a) it creates several `User` objects, each with a unique email address; (b) it creates an `AddrBook` object,

and passes to it a list of pairs (name,user), `[p1,...]`; (c) it creates a `Notifier` object which receives the address book `ab` as class parameter; (d) it adds some email addresses to be notified by asynchronously calling `addUsers`, and waits until it has terminated; and (e) finally it calls method `notify` in order to notify all registered users with a given message.

```
User u1=new UserImp();
Pair<String,User> p1 =Pair("John",u1);
...
AddrBook ab=new AddrBook(map[p1, p2, p3]);
MailServer ms =new MailServer(ab);
Fut<Unit> x =ms!addUsers(list["Alice","Bob"]);
await x?;
ms!notify("Hello Alice and Bob");
```

Fig. 6. Usage scenario: Main method

First, the MHP analysis generates a MHP graph that captures all MHP relations between the different program points of the program. Then, using this graph, it infers the set of MHP pairs of the form (i,j) which indicates that the instruction at program point i might execute in parallel with the one at program point j, and vice versa.

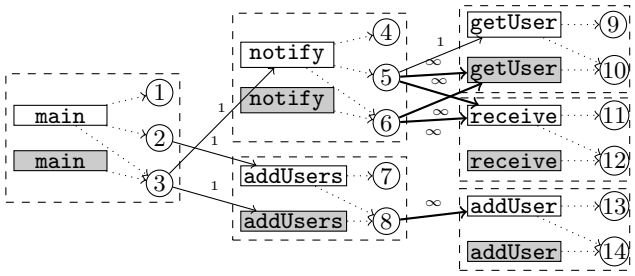


Fig. 7. MHP graph

MHP Graphs. The MHP graph corresponding to our current example is depicted in Fig. 7. Each program point i that corresponds to a context switch, i.e., a program point in which the execution might switch from one method to another, is represented by a node (i). These nodes always include the method's *entry* and *exit* program points. In principle, other program points can be included, however, these are the only ones required for soundness. Each method `m` contributes two nodes: `m` represents an instance of `m` that is *active*, i.e., running

and can be at any program point, and m represents an instance of m that is *finished*, i.e., it is at the exit program point.

The MHP graph is composed of 6 subgraphs, one for each method and that are represented as dashed rectangles. In each subgraph: (a) the *active* method node (the white rectangle) is connected to all program point nodes of that method, meaning that when the method is active it can be executing at any of those program points; and (b) the *finished* method node (the gray rectangle) is connected to the exit program point node, meaning that when the method is finished it must be at the exit program point. For example, in the subgraph of method main, there are edges from main to nodes ①, ②, and ③; and from main to ③.

The subgraphs are interconnected by weighted edges. Each such edge starts at a program point node in one subgraph, and ends in an active or finished method node in another subgraph (it can be the same if the method is recursive). These edges are inferred by applying a *method-level* MHP analysis which analyzes each method separately. This analysis infers, for each program point, which methods might be running in parallel with that program point, how many instances of each, and in which mode (active or finished). This information is inferred by considering only the code of the corresponding method. For example, the method-level analysis infers: (a) for method main, at 2 (that corresponds to the *await* instruction), there might be one active instance of method addUsers. This will add an edge from ② to addUsers. The edge is labeled with 1 to indicate that it is only one instance of addUsers; and (b) for method notify, at 5(the *await* instruction), there might be an active instance of getUser, many finished instances of getUser, and many active instances of receive. This will add an edge from ⑤ to getUser with label 1, to getUser with label ∞, and to receive with label ∞. Edges with ∞ should be interpreted as infinitely many edges with weight 1.

MHP Property. The MHP graph guarantees that if there is an execution in which the instructions at program points i and j might execute in parallel, at least one of the following holds:

- *direct relation*: there is a path from (i) to (j) (or vice versa); or
- *indirect relation*: there is a node (k) that has two *different* paths to both (i) to (j).

These properties are the base of the MHP inference. We can see that there is a path from ② to ⑬ which induces the direct MHP pair (2 ,13). Also, there are different paths from ③ to both ⑬ and ⑤ which induces the indirect MHP pair (5,13). Given this pair we cannot verify our invariant. In fact, we just detected a synchronization error. At 5(where our invariant is placed) the *emails* list can be modified by the method addUser.

The MHP analysis can also be used to spot synchronization errors, find the causes of those errors (debugging), and acquire a better understanding of the program concurrent behavior (program understanding) –See [11,12].

5 Advanced Topics in Resource Analysis

We briefly overview some advanced topics in resource analysis which we have investigated within the context of the COSTA system.

5.1 Memory Consumption Analysis

Predicting the dynamic memory (heap) required to run a program is crucial in many contexts such as in embedded applications with stringent space requirements or in real-time systems which must respond to events or signals within a predefined amount of time. On the other hand, garbage collection (GC) is a very powerful and useful mechanism which is increasingly used in high-level languages such as Java. Unfortunately, GC makes it difficult to predict the amount of memory required to run a program. A first approximation to this problem is to simply ignore the GC and infer bounds on the *total heap consumption*, i.e., the *accumulated* amount of memory dynamically allocated by a program. This can be done directly applying the COSTA framework using the $\mathcal{M}_h$ cost model defined in Section 2. If such amount of memory is available it is ensured that the program can be executed without exhausting the memory, even if no GC is performed during its execution. However, this is an overly pessimistic estimation of the actual heap consumption since, in the presence of GC, the memory usage increases and decreases along the execution.

COSTA incorporates a novel *peak heap space analysis* [13], also known as *live heap space analysis*, which aims at approximating the maximum size of the data on the heap during a program's execution, which provides a much tighter estimation. Whereas analyzing the total heap consumption requires to observe the consumption at the *final* state only, peak heap consumption analysis has to reason on the heap consumption at *all program states* along the execution. As a consequence, the basic COSTA framework cannot be directly applied.

When considering GC, several techniques exist which differ on:

(1) *what* can be collected, i.e., the *lifetime* of objects;
(2) *when* GC is performed.

As regards (1), a *GC strategy* classifies objects in the heap into two categories: those which are collectible and those which are not. Most types of garbage collectors determine *unreachable* objects as collectible, i.e., they eliminate those objects to which there is no variable in the program environment pointing directly or indirectly. The more precise alternative is to rely on the notion of *liveness*. An object is said to be *not live* (or *dead*) at some state if it is not used from that point on during the execution.

As regards (2), we consider several possibilities. One is *scope-based* GC in which deallocation of unreachable objects takes place on methods' return, and, only those objects created during the method's execution can be freed. Another possibility is the so-called *ideal* GC in which objects are collected as soon as they become collectible. The third one assumes a given limit on the heap, and applies GC only when we are about to exceed this limit.

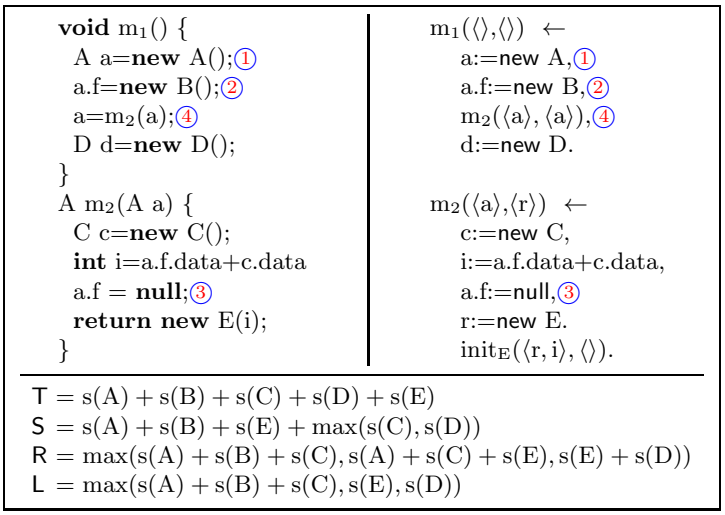


Fig. 8. A Java Program and its memory requirements: T=total-allocation; S=scope-based; R=reachability-based; L=liveness-based.

COSTA offers a general framework to infer accurate bounds on the peak heap consumption of bytecode programs which improves the state-of-the-art in that:

- it is not restricted to any complexity class and deals with all bytecode language features including recursion,
- it is parametric w.r.t the *lifetime* of objects and,
- it can be instantiated with different GC strategies, e.g., the scope-based and ideal GC discussed above.

Example 7. Let us consider the Java program in Figure 8 (to the left). To the right, we show the RBR. Because the program has simple (constant) memory consumption, it is useful to describe intuitively the differences among the different approximations to memory consumption. In Figure 8 (to the bottom) we provide four possible approximations inferred by our analysis for the memory consumption of executing method m_1, where the notation s(X) means the memory required to hold an instance of class X.

First, we consider a scope-based GC in which objects lifetimes are inferred by an *escape* analysis. In this case, we can take advantage of the knowledge that at ④ (i.e., upon exit from m_2) the object to which "c" refers can be freed, i.e., it does not *escape* from the method. Hence, the UB S is obtained. The important point is that s(A) and s(B) are always accumulated, plus the largest of the consumption of m_2 (i.e., s(C) + s(E)) and the memory *escaped* from m_2 (i.e., s(E)) plus the continuation (i.e., s(D)).

As another instance, we consider a reachability-based GC but without the assumption of being scope-based, rather we assume an ideal GC. Then, our

method is able to obtain the UB R in Fig 8. This is due to the fact that the object to which "a.f" points becomes unreachable at program point ③, the object to which "c" points becomes unreachable upon exit from m_2, and the object created immediately before ① becomes unreachable at ④. We can observe that this information is reflected in R by taking the maximum between: the consumption up to the first allocation instruction in m_2; the consumption up to the end of m_2 taking into account that the object to which "a.f" points becomes unreachable, plus the consumption until the end of m_1 taking into account that both the object pointed by "a.f" and the object created immediately before ① become unreachable.

As the third instance, we consider the combination of an ideal garbage collector based on liveness, i.e., objects are reclaimed as soon as they become dead (i.e, will not be used in the future). Then, we obtain the UB L by taking advantage of the fact that the object created immediately before ① and those to which "a.f" and "c" point are dead at program point ③, and that the object created at the end of m_2 is dead at program point ④. This information is reflected in the elements of the max similarly to what we have seen for R. Note that, in theory, the peak heap consumption L is indeed the minimal memory requirement for executing the method.

5.2 Inference of Task-Level in Concurrent Languages

Another type of non-cumulative resource is the task-level. In parallel languages, we refer to a *task* as the unit of parallelism in a program execution, i.e., a sequential computation which can be executed in parallel and communicate with a number of other computations going on at the same time. The *task level* of a program is the maximum number of tasks that can be *available* (i.e., not finished nor suspended) simultaneously during its execution, regardless of the input data (e.g., considering all possible inputs). Knowing statically the task level of a program is of utmost importance for program understanding, debugging, and task scheduling.

COSTA includes a component which estimates the task level of parallel programs written in a subset of the X10 programming language. X10 features *async-finish parallelism*, where `async` and `finish` are basic constructs for, respectively, spawning a new task and waiting until some tasks terminate. In particular `finish` waits until all tasks spawned within the block that it delimits finish. Given a parallel program, our analysis [9] returns a *task-level upper bound*, i.e., a function on the input arguments that guarantees that the task level of the program will never exceed its value along any execution.

Example 8. The following recursive program implements the *merge-sort* algorithm working on a global array, and shows how `async-finish` parallelism works:

```
void msort(int from, int to) {
  if (from<to) {
    mid=(from+to)/2;
```

```
    finish {
      async msort(from,mid);
      async msort(mid+1,to);
    }
    merge(from,to,mid);
  } }
```

`msort` recursively calls itself twice inside `async`: this means that the tasks `msort(from,mid)` and `msort(mid+1,to)` are spawned asynchronously and can execute independently. However, calling them inside `finish` means that the main procedure cannot continue to `merge(from,to,mid)` until both tasks have finished.

In the above example, the *total number of tasks* (i.e., all the tasks which may be spawned along the execution) spawned by a call to `msort(from,to)` is bounded by $2(\texttt{to}-\texttt{from}+1)-2$. Moreover, it can also be inferred that the *peak of live tasks* (i.e., the maximum number of tasks which can be alive at the same time) is bounded by the same expression. In general, the peak of live tasks is smaller than the total number since it is often the case that tasks are created only after other tasks have finished (this can be implemented by using `finish`).

Both notions of task level discussed above disregard whether a created task is *active*, i.e., actually executing, or *suspended*. In the example, the main procedure is suspended while the execution of the `finish` block is going on. Our analysis can give an upper bound $\texttt{to}-\texttt{from}+1$ to the *peak of available* tasks (i.e., those which can be active at the same time), which is almost half the one obtained for total and live tasks. Such bound is useful when allocating tasks on processors, since active tasks are the only ones actually needing a processor to execute.

The analysis uses most of the machinery already developed in COSTA, and defines all the notions of task level as something similar to cost models. Further effort has been devoted to adapting the analyzer to this specific language, and to optimizations: for example, knowing which tasks, among the ones spawned inside a procedure p, are still live after p ends allows, in general, improving the obtained upper bounds.

5.3 Heap-Sensitive Analysis

Shared mutable data structures, such as those stored in the heap, are the bane of formal reasoning and static analysis. In object-oriented programs most data reside in objects and arrays stored in the heap. Analyses which keep track of heap-allocated data are referred to as *heap-sensitive*. Most existing value analyses are only applicable to numeric variables which satisfy two *locality* conditions: (1) all occurrences of a variable refer to the same memory location, and (2) memory locations can only be modified using the corresponding variable. The conditions above are not satisfied when numeric variables are stored in shared mutable data structures such as the heap. Condition (1) does not hold because memory locations (numeric variables) are accessed using reference variables, whose value can change during the execution. Condition (2) does not hold because a memory

location can be modified using different references which are aliases and point to such memory location.

Example 9. Consider the following loops where f is a field of integer type:

```
①while(x.f > 0) {
  x.f = x.f-1;
}
```

```
②while(x.f > 0) {
  x.f = x.f-1;
  x = x.next;
}
```

```
③while(x.y.f > 0) {
  x.f = x.f-1;
  y.f = y.f+1;
}
```

Loop ① terminates in sequential execution because x.f decreases at each iteration and, for any initial value of x.f, there are only a finite number of values which x.f can take before reaching zero. Unfortunately, applying standard value analyses on numeric fields can produce wrong results, and further conditions are required. E.g., if we add the instruction x=x.next; as in loop ②, the memory location pointed to by x.f changes, invalidating Condition 1. Also, Condition 2 is false if we add y.f=y.f+1 as in loop ③, as x and y may be aliases.

The approach developed in COSTA [15,7,5] is based on the observation that, by analyzing scopes, rather than the application as a whole, it is often possible to keep track of heap accesses in a similar way to local variables. Our approach consists of the following steps: (1) partition the program to be analyzed into scopes, in our case we use the *strongly connected components* of the program, (2) identify with a *reference constancy analysis* the *access paths* used to access to the heap, (3) check the above *locality* conditions for the access paths, that is, if an access path meets the above conditions it can be safely handled by a heap-insensitive analysis, (4) transform the program by introducing local *ghost* variables whose values represent the values of the corresponding heap accesses, and (5) analyze the transformed program scope by scope using any heap-insensitive static analyzer, in particular, the one that we already have in COSTA. Let us describe the main components of the heap-sensitive analysis implemented in COSTA:

Reference Constancy Analysis. We first develop a *reference constancy analysis* which infers those memory locations of fields which are constant in the considered scope. The idea behind this analysis is similar in spirit to that of the classical numeric constant propagation analysis [20]. However, in addition to numerical constants, the values computed by our analysis can include symbolic expressions that refer to locations in (the initial) heap. Such expressions encode as well the way in that the corresponding memory locations are reached (e.g., the dereferenced fields). Intuitively, our analysis will assign to each variable (at each program point) an *access path* which describes its possible memory locations whenever the execution reaches that point. Access paths can be separated in *read* accesses and *write* accesses. In the analysis, if a reference variable is updated inside a scope its access path is not constant in this scope, so we cannot keep track of this variable by using a local variable.

Example 10. Let us consider the loops in Ex. 9. We obtain the following read $R(f)$ and write sets $W(f)$ for field f by applying the reference constancy analysis.

①$R(f) = \{x.f\}$ $W(f) = \{x.f\}$ ②$R(f) = \emptyset$ $W(f) = \emptyset$ ③$R(f) = \{x.f, y.f\}$ $W(f) = \{x.f, y.f\}$

Observe that in loop ① field f is accessed only using reference variable x. In loop ② the sets are empty because the location of variable x is updated by the instruction x = x.next inside the loop, so it is not constant in the scope of the loop. Loop ③ has two different access paths through x and y to field f.

Locality. Intuitively, in order to ensure a sound transformation, a field f can be considered local in a scope S if all read and write accesses to it in all reachable scopes are performed through the same access path l, that is, if $R(f) \cup W(f) = \{l\}$. This makes it safe to replace such heap access by a corresponding local ghost variable.

Example 11. From the results obtained in Ex. 10, and according to the locality condition above, field accesses in loop ① meet the locality condition since $R(f) \cup W(f) = \{\text{x.f}\}$. Field f is not local in the scope of loops ② neither ③, in ② because the union $R(f) \cup W(f)$ is empty, and in ③, the union contains two different references accessing field f, $\{x.f, y.f\}$.

Transformation. Our approach is based on instrumenting the program with extra local (ghost) variables that expose the values of those locations to a heap-insensitive analysis as follows: (1) for each heap access that is local in scope S, we introduce a ghost variable g; (2) when the content of the memory location is modified, we modify g accordingly; and (3) when the memory location is read, we read the value from g. This approach has one clear advantage: there is no need to change existing static analysis tools to make them heap-sensitive, we simply apply them on the transformed program, and then the properties inferred for the ghost variables hold also for the corresponding memory locations.

Example 12. Field f behaves locally in loop ①, so the heap accesses to f can be transformed into ghost variables resulting in the following program, whose termination and cost can be inferred by a heap-insensitive analyzer:

```
g = x.f;
while(g > 0) {
  g = g-1;
}
x.f = g;
```

Recently, COSTA has included an abstract interpretation based heap-sensitive analysis [44] that infers reachability and acyclicity of heap allocated data structures. This analysis infers weather some reference variables point to an acyclic data structure which is useful for the analysis of termination and resource usage.

5.4 Incremental Resource Analysis

A key challenge for static analysis techniques is achieving a satisfactory combination of precision and scalability. Making precise (and hence expensive) static

analysis incremental is a step forward in this direction. The difficulty when devising an incremental analysis framework is to recompute the least possible information and do it in the most efficient way. Incremental resource usage analysis comprises two phases: (1) devising an incremental analysis framework for all pre-analysis required to infer cost relations and (2) making the process of computing a closed-form upper bound incremental.

As regards (1), in other approaches to incremental analysis, such as in the logic programming context [25], the analysis is focused in a single abstract domain. In contrast, COSTA includes a *multi-domain* incremental analysis engine [10] which can be used by all global pre-analyses required to infer the resource usage of a program (including class analysis, sharing, cyclicity, constancy and size analysis). The engine is multi-domain in the sense that it interleaves the computation for the different domains and takes into account dependencies among them, in such a way that it is possible to invalidate only partial pre-computed information.

The incremental analysis engine starts from a program, its analysis results, and a modified method m. In addition to reanalyzing method m for all domains, other methods may require reanalysis as well: namely methods invoked by m with a different calling description (those methods are the *descendants* of m in the program call-graph); and methods which invoke m and, because of new results, require their reanalysis (referred to as *ancestors* of m). The process of incremental reanalysis transitively reanalyzes descendants and ancestors of previously reanalyzed methods, until a fixed point is reached. Once the new pre-analyses results have been computed, cost relations that correspond to reanalyzed methods are recomputed.

As regards (2), an upper bound is a global expression which includes the upper bounds of the relations it calls. If method m changes, the upper bound expressions that (directly or transitively) use m are no longer valid, since it is not possible to distinguish within an upper bound which part of the cost is associated to m. A fundamental idea to support incremental inference of upper bounds is to annotate each cost subexpression with the name of the relation it comes from. For this purpose, we define the notion of *upper bound summary* to keep the annotated cost expression, the invariant and the size relations for a method m. Given an upper bound summary for a method, it is possible to replace the cost subexpressions associated to those methods invoked from it whose upper bounds have changed by their new upper bounds, and without having to recompute the whole upper bound for the method.

6 Conclusions and Future Work

We have overviewed the main techniques used to infer resource consumption bounds in the COSTA system. In the future, we plan to extend our work along the following directions.

Improvements in Computing Symbolic Bounds. We plan to study new techniques to infer more precise lower/upper bounds on the number of iterations that loops

perform. As this is an independent component, COSTA will directly be benefited from any improvement in this regard. In addition, so far we have used linear invariants for inferring linear ranking functions, minimum number of iterations of a loop and maximization or minimization of cost expressions. Another extension of our work would be inferring nonlinear loop invariants using symbolic summation and algebraic techniques. Another possible direction is inferring nonlinear input-output (size) relations for methods by viewing the output as the cost that is consumed by the corresponding method. This way, we can view the problem of inferring such input-output relations as solving corresponding CR. Also, we plan to develop new techniques to solve cost relations, to handle some programs for which amortised analysis is not needed.

Acyclicity Analysis. COSTA contains a component which performs an acyclicity analysis [23] based on tracking the reachability between locations in the heap. Future work includes improving this analysis by considering, for a path between two heap locations, the name of the fields involved. For example, this could allow to detect that, in a double-linked list, cycles must forcefully traverse both the `next` and the `prev` field, so that a loop where the data structure is traversed by `x=x.next` is guaranteed to terminate in spite of the cyclicity of the data structure, since only `next` is traversed.

Heap-sensitive Analysis. The heap-sensitive approach contained in COSTA shows that analyzing program fragments rather than the application as a whole, and under certain locality conditions, it is feasible to keep track of heap-allocated data by means of local variables. However, there are cases when the locality conditions cannot be proven unconditionally. In such cases, we seek to provide *aliasing preconditions* between the input arguments which, when satisfied in the initial state, can guarantee the termination of the program.

Proving Termination of Concurrent Programs. In the current COSTA system, termination is independently proved for each concurrent component. That is, no assumptions are made about the interactions between different concurrent components. Unfortunately, this approach is insufficient for many real world applications where several concurrent tasks depend on each other. We plan to investigate these cases in the future. Another possible research line is to consider conditional termination in open systems. In this case, we should extract under which conditions a given component terminates.

Deadlock Analysis of ABS Programs. Deadlocks are one of the most common errors in concurrent programs. There has been some theoretical research about deadlocks in ABS programs [24]. However, there is no practical analysis that can be used for real programs. Deadlocks can occur when several concurrent components are waiting for each other (Circular dependency). We intend to develop a deadlock analysis that combines the MHP analysis and a points-to analysis. The MHP information can detect sets of synchronization instructions that may happen in parallel and the points-to analysis can identify dependencies (which components are waiting for which others) within these sets.

Cost Analysis of Concurrent Java programs. At present, the extension of COSTA handles concurrent primitives of ABS programs where the number of context switches among concurrently running objects are determined by the release points defined by the high level language constructs. That means the concurrent code runs sequentially between two release points which simplifies the resource usage analysis for ABS programs. However, in thread-based concurrent java programs, the context switch can happen at any program point. Hence, existing analysis looses too much precision in inferring size relations or bounding loop iterations and thus does not provide any useful information on resource usage of concurrent java programs. In order to extend our analysis to handle thread-based concurrent primitives, it is essential to infer relational invariants both on shared variables and thread local variables at every program points. It is also essential to infer relations between shared and thread local variables. Thus, these invariants can be used to infer the size relations which capture the cost of the programs.

Acknowledgments. This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624, TIN2012-38137-C02 and PRI-AIBDE-2011-0900 projects, by UCM-BSCH-GR35/10-A-910502 grant and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project.

References

1. Adachi, A., Kasai, T., Moriya, E.: A Theoretical Study of the Time Analysis of Programs. In: Becvar, J. (ed.) MFCS 1979. LNCS, vol. 74, pp. 201–207. Springer, Heidelberg (1979)
2. Albert, E., Arenas, P., Alonso, D., Genaim, S., Puebla, G.: Asymptotic Resource Usage Bounds. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 294–310. Springer, Heidelberg (2009)
3. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: Cost Analysis of Concurrent OO programs. In: Yang, H. (ed.) APLAS 2011. LNCS, vol. 7078, pp. 238–254. Springer, Heidelberg (2011)
4. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: COSTABS: A Cost and Termination Analyzer for ABS. In: Procs. of PEPM 2012, pp. 151–154. ACM Press (2012)
5. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Field-Sensitive Value Analysis by Field-Insensitive Analysis. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 370–386. Springer, Heidelberg (2009)
6. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-Form Upper Bounds in Static Cost Analysis. Journal of Automated Reasoning 46(2), 161–203 (2011)
7. Albert, E., Arenas, P., Genaim, S., Puebla, G., Ramírez Deantes, D.V.: From Object Fields to Local Variables: a Practical Approach to Field-Sensitive Analysis. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 100–116. Springer, Heidelberg (2010)

8. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Object-Oriented Bytecode Programs. Theoretical Computer Science (Special Issue on Quantitative Aspects of Programming Languages) 413(1), 142–159 (2012)
9. Albert, E., Arenas, P., Genaim, S., Zanardini, D.: Task-Level Analysis for a Language with Async-Finish parallelism. In: Proc. of LCTES 2011, pp. 21–30. ACM Press (2011)
10. Albert, E., Correas, J., Puebla, G., Román-Díez, G.: Incremental Resource Usage Analysis. In: Procs. of PEPM 2012, pp. 25–34. ACM Press (2012)
11. Albert, E., Flores-Montoya, A.E., Genaim, S.: Analysis of May-Happen-in-Parallel in Concurrent Objects. In: Giese, H., Rosu, G. (eds.) FORTE 2012 and FMOODS 2012. LNCS, vol. 7273, pp. 35–51. Springer, Heidelberg (2012)
12. Albert, E., Flores-Montoya, A., Genaim, S.: Maypar: A May-Happen-in-Parallel Analyzer for Concurrent Objects. In: Proc. of FSE-20, SIGSOFT/FSE 2012, pp. 1–4. ACM (2012)
13. Albert, E., Genaim, S., Gómez-Zamalloa, M.: Heap Space Analysis for Garbage Collected Languages. Science of Computer Programming (2012) (to appear)
14. Albert, E., Genaim, S., Masud, A.N.: More Precise yet Widely Applicable Cost Analysis. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 38–53. Springer, Heidelberg (2011)
15. Albert, E., Genaim, S., Román-Díez, G.: Conditional Termination of Loops over Arrays. In: Proc. of Bytecode 2012 (2012)
16. Alonso-Blas, D.E., Genaim, S.: On the Limits of the Classical Approach to Cost Analysis. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 405–421. Springer, Heidelberg (2012)
17. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. Science of Computer Programming 72(1-2), 3–21 (2008)
18. Benzinger, R.: Automated Higher-Order Complexity Analysis. Theoretical Computer Science 318(1-2), 79–103 (2004)
19. Bruynooghe, M., Codish, M., Gallagher, J.P., Genaim, S., Vanhoof, W.: Termination analysis of logic programs through combination of type-based norms. ACM Transactions on Programming Languages and Systems 29(2) (2007)
20. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proc. of POPL 1977, pp. 238–252. ACM Press (1977)
21. Debray, S.K., Lin, N.W.: Cost Analysis of Logic Programs. ACM Transactions on Programming Languages and Systems 15(5), 826–875 (1993)
22. Feautrier, P.: Parametric Integer Programming. RAIRO Recherche Opérationnelle 22(3), 243–268 (1988)
23. Genaim, S., Zanardini, D.: The Acyclicity Inference of COSTA. In: 11th International Workshop on Termination (2010)
24. Giachino, E., Laneve, C.: Analysis of Deadlocks in Object Groups. In: Bruni, R., Dingel, J. (eds.) FORTE 2011 and FMOODS 2011. LNCS, vol. 6722, pp. 168–182. Springer, Heidelberg (2011)
25. Hermenegildo, M., Puebla, G., Marriott, K., Stuckey, P.: Incremental Analysis of Constraint Logic Programs. ACM Transactions on Programming Languages and Systems 22(2), 187–223 (2000)
26. Hermenegildo, M.V., Bueno, F., Carro, M., López, P., Mera, E., Morales, J.F., Puebla, G.: An Overview of Ciao and its Design Philosophy. Theory and Practice of Logic Programming 12(1-2), 219–252 (2012), http://arxiv.org/abs/1102.5497

27. Hoffmann, J., Aehlig, K., Hofmannn, M.: Multivariate Amortized Resource Analysis. ACM Transactions on Programming Languages and Systems 34(3), 14:1–14:62 (2012)
28. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A Core Language for Abstract Behavioral Specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
29. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall (1993)
30. Jost, S.: Automated Amortised Analysis. PhD thesis, Ludwig-Maximilians-Universität (August. 2010)
31. Le Metayer, D.: ACE: An Automatic Complexity Evaluator. ACM Transactions on Programming Languages and Systems 10(2), 248–266 (1988)
32. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley (1996)
33. Navas, J., Mera, E., López-García, P., Hermenegildo, M.V.: User-Definable Resource Bounds Analysis for Logic Programs. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 348–363. Springer, Heidelberg (2007)
34. Podelski, A., Rybalchenko, A.: A complete Method for the Synthesis of Linear Ranking Functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
35. Rodriguez, D.: Amortised Resource Analysis for Object-Oriented Programs. Phd thesis, LMU Munich (October 2012)
36. Rosendahl, M.: Automatic Complexity Analysis. In: Proc. of FPCA 1989, pp. 144–156. ACM Press (1989)
37. Sands, D.: A Naïve Time Analysis and its Theory of Cost Equivalence. Journal of Logic and Computation 5(4), 495–541 (1995)
38. Schäfer, J., Poetzsch-Heffter, A.: Jcobox: Generalizing Active Objects to Concurrent Components. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 275–299. Springer, Heidelberg (2010)
39. Srinivasan, S., Mycroft, A.: Kilim: Isolation-Typed Actors for Java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 104–128. Springer, Heidelberg (2008)
40. Tarjan, R.E.: Amortized Computational Complexity. SIAM Journal on Algebraic and Discrete Methods 6(2), 306–318 (1985)
41. Wadler, P.: Strictness Analysis Aids Time Analysis. In: ACM Symposium on Principles of Programming Languages (POPL 1988). ACM Press (1988)
42. Wegbreit, B.: Mechanical Program Analysis. Communications of the ACM 18(9), 528–539 (1975)
43. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming 12(1-2), 67–96 (2012)
44. Genaim, S., Zanardini, D.: Reachability-based Acyclicity Analysis by Abstract Interpretation. Theoretical Computer Science (2013)
45. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-Form Upper Bounds in Static Cost Analysis. Journal of Automated Reasoning 46(2), 161–203 (2011)
46. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: COSTA: Design and implementation of a cost and termination analyzer for java bytecode. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 113–132. Springer, Heidelberg (2008)

Separating Cost and Capacity for Load Balancing in ABS Deployment Models *

Einar Broch Johnsen

Department of Informatics, University of Oslo, Norway
einarj@ifi.uio.no

Abstract. Software is often developed for a range of deployment scenarios; different versions of the software may be specialized for a number of distributed and even virtualized architectures. Since software performance can vary significantly depending on the target architecture, design decisions may need to address which features to include and what performance to expect for the different architectures. If the load of the software system depends on external parameters (such as users), the software may also need to include dynamic load balancing strategies to alleviate congestion and thereby improve its own performance.

Executable models in the abstract behavioral specification language ABS can support such design decisions by explicitly modeling deployment scenarios, including load, congestion, response time, etc. This paper gives an overview of how deployment scenarios can be captured in ABS. A separation of concerns between execution cost at the object level and execution capacity at the deployment level makes it easy to compare timing and performance for different deployment scenarios early in system modeling. The language and associated simulation tool is demonstrated on an example of a virtual world framework for distributed gaming.

1 Introduction

This paper introduces deployment modeling in the abstract behavioral specification language ABS by examples. ABS aims at high-level models that abstract from implementation details but captures essential behavioral aspects of the targeted systems [20]. ABS supports the design of concurrent, component-based systems by means of executable object-oriented models which are easy to undertand for the software developer and allow rapid prototyping and analysis.

To express and compare deployment decisions at the modeling level, ABS has been extended with *deployment components* [23]. Whereas software components reflect the logical architecture of systems, deployment components reflect their deployment architecture. The deployment architecture expresses how computing units are statically or dynamically mapped to virtual or physical locations where execution occurs. A deployment component is a context with a given execution

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

E. Giachino et al. (Eds.): FMCO 2012, LNCS 7866, pp. 145–167, 2013.

capacity which restricts how much computation can occur in an accounting period. Individual statements in ABS have an associated execution *cost*. By separating cost and capacity, the time needed to perform an execution will depend on where the execution occurs; for example, selecting a server with higher capacity will normally (but not always) decrease the computation time.

ABS is a modeling language based on concurrent object groups which execute in parallel and communicate by asynchronous method calls. Inside an object group, different tasks may be executed in an interleaved way by means of cooperative scheduling [21]. This allows an object to temporarily suspend its active task and pursue another task, for example while the first task is waiting for a reply from another object. In this setting, deployment components may be dynamically created in the same way as objects, and they are parametric in the resource capacity they provide to their set of concurrent object groups. This explicit representation of deployment architectures allows application-level response time and load balancing to be expressed in the system models in a very natural and flexible way, relative to the resources allocated to an application.

Although this paper is informal and example-driven, ABS and its extension to real-time and deployment models are formally defined specification languages which have been detailed in a number of papers. For further details of deployment modeling in ABS, including the formalization and other complementary examples, the reader is referred to the following papers: The formal syntax and semantics of ABS is presented in [20], the extension to real-time in [7], and the formal explanation of how deployment components can be integrated with ABS has been studied in [4, 22–25]. A paper at FMCO 2011 proposed an ABS model for cloud provisioning [27], based on deployment components. This library has been used in two larger case studies of virtualized computing on the cloud [12,26].

This paper illustrates the use of deployment components in ABS by developing an example which models congestion in distributed online games, inspired by a framework for virtual worlds [30]. In this example, players move their avatars between rooms and collect artifacts they find in these rooms. The actions of avatars have deadlines, expressing the quality of service expected by the players. When an action is performed, its cost depends on a weight associated with the involved artifacts. The rooms are deployed on deployment components with given capacities that we use to measure the congestion in different rooms. We then show how load balancing may be introduced in this scenario in a way which is orthogonal to the behavioral specification of the virtual worlds.

The paper is structured as follows. Section 2 introduces ABS and shows how to express, e.g., deadlines in ABS models. Section 3 presents the running example of virtual worlds and a behavioral model for the example. Section 4 introduces the modeling of deployment architecture in terms of deployment components with resource capacities and code annotated with resource cost. Section 5 considers how load balancing in the deployment architecture can be controlled by the behavioral model, and integrates load balancing in the virtual worlds example. Section 6 discusses related work and Sect. 7 concludes the paper.

2 ABS: Abstract Behavioral Specification

ABS [20] is an executable object-oriented modeling language which targets distributed systems. It is based on concurrent object groups, akin to concurrent objects (e.g., [8,11,21]), Actors (e.g., [1,18]), and Erlang processes [5]. In ABS, concurrent object groups execute processes which stem from method activations. A characteristic feature of concurrent object groups in ABS is that they internally support interleaved concurrency of processes by means of guarded commands. This allows active and reactive behavior to be easily combined in the concurrent object groups, based on cooperative scheduling of processes. A concurrent object group has at most one active process at any time and a queue of suspended processes waiting to execute on an object in the group. Objects in ABS are dynamically created from classes but typed by interface; i.e., there is no explicit notion of hiding as the object state is always encapsulated behind interfaces which offer methods to the environment. For simplicity in this paper, we do not consider other code structuring mechanisms, such as deltas [10]. This section informally reviews the core ABS language and its timed extension (for technical details, see [7,20]).

2.1 The Layers of ABS

ABS combines functional and imperative modeling layers to develop high-level executable models. Concurrent object groups execute in parallel and communicate by asynchronous method calls. To intuitively capture internal computation inside a method, ABS offers a simple functional layer 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, and still maintain an overall object-oriented design and execution flow which is close to the target system. At a high level of abstraction, concurrent object groups typically consist of a single concurrent object; other objects may be introduced into a group as required to give some of the algebraic data structures an explicit imperative representation in the model.

The functional layer of ABS consists of algebraic data types such as the empty type Unit, booleans Bool, integers Int, parametric data types such as sets Set<A> and maps Map<A> (given a value for the type variable A), and (parametric) functions over values of these data types.

Example 1. Polymorphic sets can be defined using a type variable A and two constructors EmptySet and Insert:

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

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

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

Here, the cases p => exp are evaluated in the listed order, underscore works as a wild card in the pattern p, and variables in p are bound in the expression exp. ABS provides a library with standard data types and functions.

Example 2. The data type DCData models CPU capacity, which may be either unrestricted, expressed by the constructor InfCPU, or restricted to r resources, expressed by the constructor CPU(r).

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

The observer function capacity is defined for the constructor CPU, such that capacity(CPU(r)) returns r. It is not defined for InfCPU.

The imperative layer of ABS addresses concurrency, communication, and synchronization at the concurrent object level in the system design, and defines interfaces and methods with a Java-like syntax. ABS objects are *active*; i.e., their run method, if defined, gets called upon creation. *Statements* are standard for sequential composition $s_1; s_2$, assignments x = rhs, and for the **skip**, **if**, **while**, and **return** constructs. The statement **suspend** unconditionally suspends the execution of the active process of an object by adding this process to the queue, from which an enabled process is then selected for execution. In **await** g, the guard g controls the suspension of the active process and consists of Boolean conditions b and return tests x? (see below). Just like functional expressions e, guards g are side-effect free. If g evaluates to false, the active process is suspended; i.e., the active process is added to the queue, and some other process from the queue may execute. *Expressions* rhs include the creation of an object group **new cog** C(e), object creation inside the group of the creator **new** C(e), method calls o!m(e) and o.m(e), future dereferencing x.**get** (see below), and pure expressions e apply functions from the functional layer to state variables.

Communication and *synchronization* are decoupled in ABS. Communication is based on asynchronous method calls, denoted by assignments f=o!m(e) to future variables f. Here, o is an object expression and e are (data value or object) expressions providing actual parameter values for the method invocation. (Local calls are written **this**!m(e).) After calling f=o!m(e), the future variable f refers to the return value of the call and the caller may proceed with its execution *without*

blocking on the method reply. There are two operations on future variables, which control synchronization in ABS. First, the guard **await** f? *suspends the active process* unless a return to the call associated with f has arrived, allowing other processes in the object group to execute. Second, the return value is retrieved by the expression f.**get**, which *blocks all execution in the object* until the return value is available. The statement sequence x=o!m(e);v=x.**get** encodes commonly used *blocking calls*, abbreviated v=o.m(e) (often referred to as synchronous calls). If the return value of a call is without interest, the call may occur directly as a statement o!m(e) with no associated future variable. This corresponds to message passing in the sense that there is no synchronization associated with the call.

The timed layer of ABS is an extension of ABS, called Real-Time ABS [7], which captures the timed behavior of ABS models. An ABS model is a model in Real-Time ABS in which execution takes zero time. Timing aspects may be added incrementally to an untimed behavioral model. Real-Time ABS extends ABS with *deadlines* and with both *explicit* and *implicit* time.

Deadlines. The object-oriented perspective on timed behavior is captured by *deadlines* to method calls. Every method activation in Real-Time ABS has an associated deadline, which decrements with the passage of time. This deadline can be accessed inside the method body using the expression **deadline**(). Deadlines are *soft*; i.e., **deadline**() may become negative but this does not in itself block the execution of the method. By default the deadline associated with a method activation is infinite, so in an untimed model deadlines will never be missed. Other deadlines may be introduced by means of call-site *annotations*.

Real-Time ABS introduces two data types into the functional layer of ABS. The data type Time has the constructor Time(r) and the accessor function timeVal which returns r for the value Time(r). The data type Duration has the constructors InfDuration and Duration(r), where r is a value of the type Rat of rational numbers, and the accessor function durationValue which returns r for the value Duration(r). For simplicity ABS does not support operator overloading, instead we can define a function scale which multiplies a duration by a rational number and a function timeDifference which compares two time values.

Example 3. The data types Time and Duration and the functions scale and timeDifference can be defined as follows:

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

def Duration scale(Rat r, Duration d)= Duration(r*durationValue(d));
def Rat timeDifference(Time t1, Time t2) = abs(timeValue(t2)−timeValue(t1));
```

Here, the function abs returns the absolute value.

Example 4. Consider a class which implements two methods m and n. Method m performs some computation depending on an input x and returns a Boolean value which indicates whether the method activation has met its provided deadline. Method n calls m on **this** and specifies a deadline for this call, given as an annotation and expressed in terms of its own deadline. Remark that if the deadline to the call to n is InfDuration, then the deadline to m will also be InfDuration and n will return true independent of the actual value provided for x.

```
Bool m(T x){ s; return deadline()>0; }

Int n (T x){
  [Deadline: scale(9/10,deadline())] Bool result = this.m(x);
  return result; }
```

Explicit Time. The execution time of a computation is captured by a *duration statement* **duration**(e1,e2) in the explicit time model of Real-Time ABS [13], expressing an execution time between the best case e1 and the worst-case e2. This is a standard approach to modeling timed behavior, in, e.g., timed automata in UPPAAL [28]. In the context of deployment modeling in ABS, these statements can be inserted into the model to capture execution time which is *independent* of how the system is deployed.

Example 5. Let f be a function defined in the functional layer of ABS, which recurs through some data structure of type T, and let size(x) be a measure of the size of this data structure. Consider a method m which takes as input such a value x and returns the result of applying f to x. Let us assume that the time needed for this computation depends on the size of x; e.g., the computation time is between a duration 1/2∗size(x) and a duration 4∗size(x). An interface I which provides the method m and a class C which implements I, including the execution time for m using the explicit time model, are specified as follows:

```
interface I { T m(T x); }
class C implements I {
  T m (T x){ duration(1/2∗size(x), 4∗size(x)); return f(x);
  }
}
```

Implicit Time. In the implicit time model of Real-Time ABS, the execution time is not specified explicitly in terms of durations, but rather *observed* (or measured) during the executing of the model. This is done by comparing clock values from a global clock, which can be read by an expression **now**() of type Time.

Example 6. Consider an interface J with a method p which, given a value of type T, returns a value of type Duration. Let a class D implement J such that p measures the time needed to call the method m above, as follows:

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

Observe that by using the implicit time model, no assumptions about execution times are specified in the model above. The execution time depends on how quickly the method call is effectuated by the called object. The execution time is simply measured during execution by comparing the time before and after making the call. As a consequence, the time needed to execute a statement with the implicit time model depends on the *capacity* of the chosen deployment architecture and on *synchronization* with (slower) objects.

3 Example: Virtual Worlds

A virtual world is a simulated environment in which agents with local state interact with each other and manipulate objects. Virtual worlds are used in, e.g., computer games and social simulations. In games, the agents (or avatars) are controlled by players who try to "win" the game. In social simulations, the agents comply with simple laws or behaviors reflecting, e.g., social norms, elections, or aspects of economics, and the purpose is to study emergent structures arising from these laws or behaviors. In frameworks for virtual worlds (e.g., [30]), a world consists of *rooms* which are connected to each other by means of *portals*. An avatar may move from one room to another if these rooms are connected by a portal which will accept the agent. In addition, the world contains *artifacts* which can be in a room and which can be in the possession of an avatar.

The number of avatars and artifacts in a room varies over time. Since avatars can interact with each other, the avatars in a room are notified when another avatar enters or leaves that room. Since avatars can discover artifacts in a room, the avatars in a room are notified when artifacts become discoverable or stop being discoverable (for example, because the artifact is picked up or dropped by another avatar). Since avatars move between rooms, bringing the artifacts in their possession along, the processing load associated with different rooms vary over time. Avatars may like to interact, potentially creating congestion in particular rooms. Thus, the quality of service of the virtual world depends on how its rooms are deployed on the available servers. If we consider a virtual world for a game, the quality of service corresponds to the time it takes for the virtual world to react to the way a player controls her avatar; e.g., the virtual world should respond to each action selected by the player within x time units.

3.1 Data Types for Artifacts and Portals

Passive objects are typically modeled in the functional layer of ABS. In the virtual world, *artifacts* are passive entities manipulated by avatars, which are modeled by a data type Artifact with a constructor Artifact. Since artifacts can be moved from one room to another by a moving avatar, each artifact has an associated *weight*. A *portal* connects the exit from one room to the entrance of another, so each portal has an associated exit which can be retrieved by the accessor function getExit:

```
data Artifact = Artifact(Int weight);
data Portal = Channel(Room getExit);
```

3.2 Interfaces for Avatars and Rooms

Avatars and rooms are the active entities in the virtual world system. An avatar virtually represents a player. A room represents the infrastructure of the virtual world, which notifies the avatars located in that room of events; e.g., an artifact can be discovered, an avatar has arrived, an avatar has left the room, etc.

The model considers two actions for an avatar: picking up an artifact in the room where the avatar is located, and moving from the current room to another room via one of the room's exits. In the Avatar interface there are two methods which can be selected by the player: moveto allows the avatar to relocate to a new room, by choosing one of the portals leading from the current room, and pickup makes the avatar acquire an artifact located in the current room.

In the model, we consider the following functionality of the infrastructure for a room with which an avatar interacts. The method getExits is used to retrieve possible exits from the room. The method getArtifact is used by an avatar to take an artifact located in the room into its possession (so it is no longer visible to other avatars in the room). The methods receiveAvatar and removeAvatar notify the framework about the arrival and departure of avatars from the room.

It is straightforward to extend the model with additional functionality usually found in virtual worlds; e.g., the ability to create new artifacts and rooms, to reconnect the rooms dynamically by moving portals, etc. To illustrate such functionality, the interface RoomAdministration provides a method connectExit which adds a new exit portal to the room. The interface FullRoom combines the two interfaces Room and RoomAdministration, so FullRoom is a subtype of both Room and RoomAdministration. The interfaces are defined in Figure 1.

3.3 Implementing the Avatar and Room Interfaces

Two classes Person and Location implement the interfaces Avatar and FullRoom, respectively. The class Player represents an active entity in the model, and will have a run method which is activated on an instance at creation time. The class Location represents the virtual world framework, and performs computations to notify players when the artifacts and players at the location change.

```
interface Avatar {
  Unit moveto(Portal door);
  Unit pickup(Artifact item);
}
interface Room {
  Set<Portal> getExits();
  Set<Artifact> getArtifact();
  Unit receiveAvatar(Avatar avatar,Int weight);
  Unit removeAvatar(Avatar avatar,Int weight);
}
interface RoomAdministration {
  Unit connectExit(Portal door);
}
interface FullRoom extends Room, RoomAdministration {}
```

Fig. 1. The interfaces for avatars and rooms

Person. The class Person, which implements the Avatar interface, is given in Figure 2. The method pickup adds an artifact to the possessions of the Person. Method moveto calculates the accumulated weight of the object and its possessions, calls the removeAvatar method of the current room so the framework can be updated, gets the new room from the requested exit, and calls the receiveAvatar method of the new room so the framework can be updated. Moving an avatar depends on its accumulated weight, which is computed by the internal method getWeight (this method is not exported through the interface of the class).

The decisions of the player selecting the actions of the artifact are simulated by the run method of the class, which is activated upon object creation. The run method is a loop which executes until the object is in the desired room target. Each iteration of the loop executes the method cycle which picks up some artifact from the current room and moves to another room. The invocation of the cycle method has limit as its deadline; this deadline represents the acceptable delay for reacting to the player's choice of actions for the avatar. Thus, the violation of this deadline models the degradation of the virtual world's quality of service. The cycle method returns false when its deadline was violated.

Location. The class Location, which implements the FullRoom interface, is given in Figure 3. It has a getter-method getExits. The method getArtifact returns an empty set if the location has no artifacts available, and otherwise a singleton set with one artifact, which is removed from the available artifacts of the location. Thus, the avatar receives all possible exits and can select one, whereas it will receive one artifact selected by the location. The methods receiveAvatar, removeAvatar, connectExit are straightforward; the parameter weight will be explained in Example 10.

```
class Person(Room originalPosition, Room target, Duration limit)
implements Avatar {
  Room position=originalPosition;
  Set<Artifact> myPossessions=EmptySet;
  Int cycles=0; Int misses=0; Bool finished = False;

  Unit pickup(Artifact item){ myPossessions=Insert(item,myPossessions);
  }
 Unit moveto(Portal door){
   Int myWeight=this.getWeight();
    await position!removeAvatar(this,myWeight);      // Leave old room
    position = getExit(door);
    duration(1,1);
    await position!receiveAvatar(this,myWeight);     // Enter new room
  }
  Int getWeight(){ Int myWeight=10;
    Set<Artifact> items=myPossessions;
    while (not(emptySet(items))){
      Artifact item = take(items); items = remove(items,item);
      myWeight=myWeight+weight(item); }
    return myWeight;
  }
  Unit run () { // Active method
    while (position != target){
     [Deadline:limit] Bool success=this.cycle();       // Response time deadline
     cycles = cycles+1;
      if (not(success)){misses=misses+1;}
    }
    finished = True;
  }
  Bool cycle(){
    Set<Artifact> availableThings = await position!getArtifact();
   if (not(emptySet(availableThings))) {
      this.pickup(take(availableThings));              // Pick up some artifact
  }
    Set<Portal> doors = await position!getExits();
    if (not(emptySet(doors))) {
        Portal door = take(doors);
        this.moveto(door);                             // Move to a new room
      }
    return (durationValue(deadline())>0);
  }
}
```

Fig. 2. The class Person, implementing the Artifact interface

```
class Location(Int amount, Int weight) implements FullRoom {
  Set<Avatar> players=EmptySet;
  Set<Artifact> stuff=EmptySet;
  Set<Portal> exits=EmptySet;

  { // Initialization code
    while (amount>0){
      stuff = Insert(Artifact(weight),stuff); amount = amount −1;
    }
  }
  Set<Portal> getExits(){ return exits; }
  Set<Artifact> getArtifact(){
    Set<Artifact> result = EmptySet;
    if (not(emptySet(stuff))){
      Artifact item = take(stuff);
      result = Insert(item,result);
      stuff=remove(stuff,item);
    }
  return result; }

  Unit receiveAvatar(Avatar avatar,Int weight){ players=Insert(avatar,players); }
  Unit removeAvatar(Avatar avatar,Int weight){ players=remove(players,avatar);}
  Unit receiveArtifact(Artifact artifact,Int weight){ stuff = Insert(artifact,stuff); }
  Unit connectExit(Portal door){ exits=Insert(door,exits); }
}
```

Fig. 3. The Rooms

4 Deployment Models in ABS

A deployment model in ABS describes the *resource capacity* of different locations where execution takes place, expressing where the executions in a specific concurrent object group will take place, and the *resource cost* of executing different parts of the behavioral model. Figure 4 illustrates a typical scenario for deployment models in ABS.

4.1 Deployment Components with Resource Capacities

A *deployment component* in Real-Time ABS captures the execution capacity associated with a number of concurrent object groups. Deployment components are first-class citizens in Real-Time ABS, and provide a given amount of resources which are shared by their allocated objects. Deployment components may be dynamically created depending on the control flow of the ABS model or

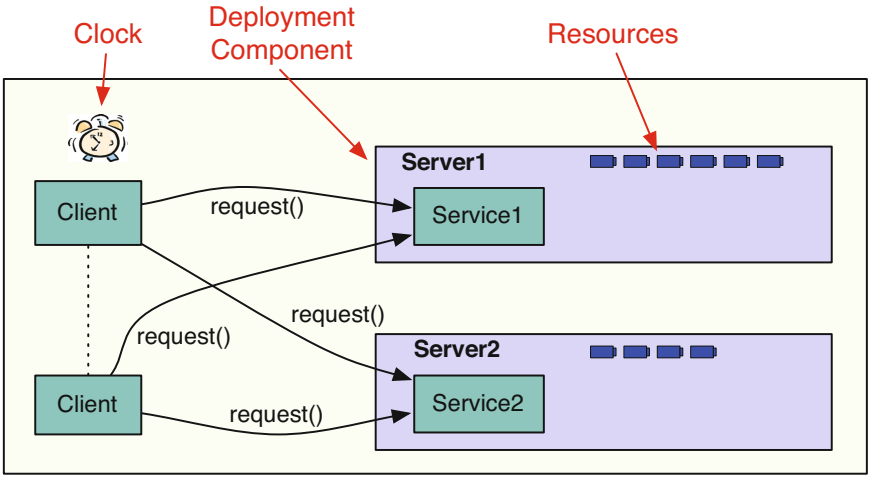


Fig. 4. Deployment models in ABS

statically created in the main block of the model. There is a deployment component environment with unlimited resources, to which the root object of a model is allocated. When objects are created they are by default allocated to the same deployment component as their creator, but they may also be allocated to a different component. Thus, for a model without explicit deployment components all execution takes place in environment, which does not impose any restrictions on the execution capacity of the model. A model may be extended with other deployment components with different processing capacities.

Deployment components in Real-Time ABS have the type DC and are instances of the class DeploymentComponent. This class takes as parameters a name, given as a string, and a set of restrictions on resources. (The name is mainly used for monitoring purposes.) Here we focus on resources reflecting the deployment components' processing capacity, which are specified in Example 2 by the constructor InfCPU for unbounded execution capacity and the constructor CPU(r) for bounded capacity, where r represents the amount of abstract processing resources available in each *accounting period*. Thus, the processing capacity is expressed for a fixed period of time, corresponding to discrete time intervals of Real-Time ABS.

Example 7. Figure 5 configures a deployment scenario for a virtual world with six rooms. First, six servers are created with a CPU capacity of 100 resources each. Then, six rooms are explicitly allocated to the servers by annotations; below, entr1 is allocated to server1, etc. Thus, each room is deployed on its own server. To connect the rooms; the method connectExit is called on each room except room4, to provide an exit via a Portal data value.

```
{ // This main block initializes a static virtual world deployment architecture:
// The deployment components
  DC server1 = new cog DeploymentComponent("entrance1server",CPU(100));
  DC server2 = new cog DeploymentComponent("entrance2server",CPU(100));
  DC server3 = new cog DeploymentComponent("room1server",CPU(100));
  DC server4 = new cog DeploymentComponent("room2server",CPU(100));
  DC server5 = new cog DeploymentComponent("room3server",CPU(100));
  DC server6 = new cog DeploymentComponent("room4server",CPU(100));

// The rooms
  [DC: server1] FullRoom entr1 = new cog Location(50,1);
  [DC: server2] FullRoom entr2 = new cog Location(50,2);
  [DC: server3] FullRoom room1 = new cog Location(40,3);
  [DC: server4] FullRoom room2 = new cog Location(30,4);
  [DC: server5] FullRoom room3 = new cog Location(50,5);
  [DC: server6] FullRoom room4 = new cog Location(10,6);

// Connecting the rooms
  entr1.connectExit(Channel(room1));
  entr2.connectExit(Channel(room2));
  room1.connectExit(Channel(room3));
  room2.connectExit(Channel(room3));
  room3.connectExit(Channel(room4));
}
```

Fig. 5. The deployment of a virtual worlds scenario

Interacting with Deployment Components. Language primitives allow objects in the behavioral model to interact with deployment components. All concurrent object groups in an ABS model are allocated to some deployment component (which is environment unless overridden by an annotation). The expression **thisDC**() evaluates to the local deployment component of a concurrent object group. Deployment components support the following method calls:

- A call to the method **total**() of a deployment component returns its total amount of allocated CPU resources;
- a call to the method **load**(n) of a deployment component returns its average load for the n last accounting periods (where n is an Integer); and
- a call to the method **transfer**(r,dc) of a deployment component transfers r of its resources to the deployment component dc.

In addition, a concurrent object group may relocate to a deployment component dc by the expression **movecogto**(dc). This primitive is not further explored in this paper; for details of its usage the reader is referred to [24].

4.2 Execution with Resource Costs

The available resource capacity of a deployment component determines how much computation may occur in the objects allocated to that deployment component. These objects compete for the shared resources of the deployment component in order to execute. They may execute until the deployment component runs out of resources or they are otherwise blocked. For CPU resources, the resources of the deployment component define its processing capacity per accounting period, after which the resources are renewed.

Cost Models. The cost of executing statements in the ABS model is determined by a cost expression for each statement. For convenience, a default value can be given as a compiler option (e.g., defaultcost=10). However, the default cost does not discriminate between statements and we often want to introduce a more refined cost model for certain parts of a model. For example, if e is a complex expression, then the statement x=e should have a significantly higher cost than **skip** in a realistic model. For this reason, more fine-grained costs can be inserted into Real-Time ABS models by means of annotations.

Example 8. Assume that the cost of computing the function f(x), which is defined in Example 5, may be given as a function g which depends on the size of the input value x. In the context of deployment components, we may redefine the implementation of interface I above to be *resource-sensitive* instead of having a predefined duration as in the explicit time model. The resulting class C2 can be defined as follows:

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

It is the responsibility of the modeler to specify an appropriate cost model. A behavioral model with default costs may be gradually refined to provide more realistic resource-sensitive behavior. For the computation of the cost functions such as g in our example above, the modeler may be assisted by the COSTABS tool [2], which computes a *worst-case approximation* of the cost for f in terms of the input value x based on static analysis techniques, when given the ABS definition of the expression f. However, the modeler may also want to capture resource consumption at a more abstract level during the early stages of system design, for example to make resource limitations explicit before a further refinement of a behavioral model. Therefore, cost annotations may be used by the modeler to abstractly represent the cost of some computation which remains to be fully specified.

Example 9. Class C3 represents a draft version of our method m in which the computation cost is specified although the function f has yet to be introduced:

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

Example 10. The class Location of the virtual world model is refined by adding a cost annotation to the methods receiveAvatar and removeAvatar. We let the resource cost of these methods correspond to the accumulated weight of the moving avatar, which again depends on the artifacts in the possession of the avatar. Here, the cost is provided by a formal parameter weight for these methods:

```
  Unit receiveAvatar(Avatar avatar,Int weight){
    [Cost: weight] players=Insert(avatar,players); }
  Unit removeAvatar(Avatar avatar,Int weight){
    [Cost: weight] players=remove(players,avatar); }
```

5 Load Balancing in ABS

Objects can interact with deployment components in ABS as described in Section 4. This allows the behavior of the ABS model to depend on the resource capacities and execution loads of its deployment component at a given point in the execution. This way, dynamic load balancing policies can be expressed and compared at the abstraction level of the modeling language. Typically, load balancing happens as follows:

- *Initial deployment decision for a concurrent object group.* The deployment component where a concurrent object group is deployed, can be decided at creation time depending on the load of available deployment components.
- *Relocation of deployed concurrent object groups.* Concurrent object groups can move between deployment components depending on the load of the deployment components or on the expected cost of an on-going or expected method activation; for example, concurrent object groups with long-running or low-priority activities may be relocated to a backup server.
- *Redistribution of resources.* Resources may be redistributed between deployment components in the setting of virtualized computing.

Other criteria may be combined with the load of the deployment components for specific load balancers; for example, a virtualized service may have a budget for how many deployment components it can acquire from a cloud provider, or it may have a quality of service requirement in terms of the acceptable percentage of missed deadlines.

Example 11. The interface LoadBalancer receives requests for resources to a deployment component *dc*. It is implemented by a class Balancer which checks

```
interface LoadBalancer { Unit requestdc(DC comp); }
class Balancer(Int limit, BalancerGroup gr) implements LoadBalancer {
  Unit run() {
    DCData total = thisDC().total(); await gr!register(this,capacity(total));
    while (True) {
      await duration(1, 1);
      total = thisDC().total(); Int ld = thisDC().load(1);
      gr.updateAvail(this, capacity(total)−ld);
      if (capacity(total) < ld ∗ limit) {
        LoadBalancer partner = gr.getPartner();
        await partner!requestdc(thisDC()); } } }
  Unit requestdc(DC comp) {
    DCData total = thisDC().total(); Int ld = thisDC().load(1);
    if (ld < (capacity(total)/2)){thisDC()!transfer(comp, capacity(total)/3);}}
}
```

Fig. 6. A load balancer for virtualized resources

```
interface BalancerGroup {
  Unit register(LoadBalancer b, Int load);
  Unit unregister(LoadBalancer b);
  Unit updateAvail(LoadBalancer b, Int w);
  LoadBalancer getPartner();
 }
class BalancerGroup() implements BalancerGroup {
  List<Pair<LoadBalancer,Int>> sorted = Nil;
  Unit register(LoadBalancer b, Int load){
    sorted = weightedInsert(Pair(b,load),sorted);}
  Unit unregister(LoadBalancer b){
    Int w = findWeight(b,sorted); sorted = without(sorted,Pair(b,w)); }
  Unit updateAvail(LoadBalancer b, Int w){this.unregister(b);this.register(b,w);}
 LoadBalancer getPartner(){
    LoadBalancer p = fst(head(sorted)); this.unregister(p); return p;}
}
```

Fig. 7. The class BalancerGroup

```
[DC: server6] BalancerGroup balGroup = new cog BalancerGroup();
[DC: server1] LoadBalancer bal1 = new cog Balancer(3,balGroup);
[DC: server2] LoadBalancer bal2 = new cog Balancer(3,balGroup);
[DC: server3] LoadBalancer bal3 = new cog Balancer(3,balGroup);
[DC: server4] LoadBalancer bal4 = new cog Balancer(3,balGroup);
[DC: server5] LoadBalancer bal5 = new cog Balancer(3,balGroup);
[DC: server6] LoadBalancer bal6 = new cog Balancer(3,balGroup);
```

Fig. 8. Load balancing the virtual world

whether there are enough available resources to comply with the request on the deployment component where it is located. The interface LoadBalancer and its implementation Balancer are given in Figure 6. The Balancer receives requests for resources for a deployment component dc through the method requestDC; If the current load on its deployment component is less than half of its capacity, it will transfer one third of its capacity to *dc*. The Balancer exchanges resources with other LoadBalancer objects via a BalancerGroup. The Balancer has a run method which monitors the load on its deployment component for every accounting period and updates its available resources with the BalancerGroup. The run method consists of a loop which is suspended for the duration of one accounting period, then it compares the current capacity and load of the deployment component. If its current capacity is less than limit $*$ **load**, it will request a LoadBalancer partner from the BalancerGroup and request additional resources from this partner.

The interface BalancerGroup allows LoadBalancer objects to register and unregister their participation in the group. Furthermore, the interface provides a method updateAvail for a LoadBalancer to update its available resources and a method getPartner to request a LoadBalancer partner. The implementation of the BalancerGroup keeps its registered LoadBalancer objects sorted by their available resources in the list sorted. The interface BalancerGroup and its implementation BalancerGroup are given in Figure 7. The function weightedInsert updates sorted when a LoadBalancer is registered, and findWeight finds the current registered available resources of a LoadBalancer. These functions are defined as follows:

```
def List<Pair<LoadBalancer,Int>> weightedInsert(Pair<LoadBalancer,Int> el,
  List<Pair<LoadBalancer,Int>> list) =
    case list {
      Nil => Cons(el,Nil);
      Cons(el2,list2) =>
        if (snd(el2) > snd (el)) then Cons(el2, weightedInsert(el,list2))
                                 else Cons(el,list); };
def Int findWeight(LoadBalancer b, List<Pair<LoadBalancer,Int>> list) =
  case list { Nil => 0;
    Cons(Pair(b,weight),_) => weight;
    Cons(_,list2) => findWeight(b,list2); };
```

The method updateAvail updates the available resources of a LoadBalancer in sorted and getPartner returns the head of this list.

5.1 Example: Load Balancing the Virtual Worlds

Load balancing is introduced into the virtual worlds example by extending the main block with one BalancerGroup as well as one LoadBalancer for each deployment component. Observe that the load balancer code is orthogonal to the behavioral code of the model: the code of the previous model has not been modified and it is straightforward to compare different load balancing strategies. The extension of the main block is given in Figure 8.

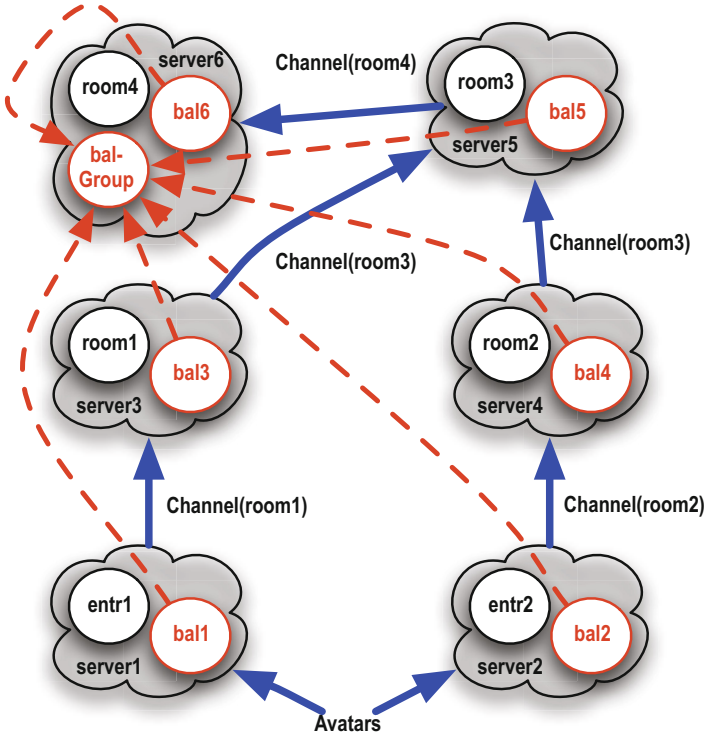


Fig. 9. The deployment model of the virtual world example

Figure 9 illustrates the virtual world model with load balancing. The dark (blue) arrows depict the Channels through which Avatar objects are moving, starting at the entrance rooms entr1 and entr2. The light (red) arrows depict the layout of the load balancing strategy, where the LoadBalancer objects deployed on the different deployment components communicate with the BalancerGroup object on deployment component server6.

To illustrate observations that can be made from executing the ABS model, we consider a scenario in which avatars with a given deadline 5 for their internal cycles, enter by the entr1 and entr2 rooms, 10 at the time. Figure 10 shows the percentage of missed deadlines for both static deployment and with added load balancing in this scenario; the vertical axis shows the percentage of missed deadlines and the horizontal axis shows the number of players (with avatars equally distributed over the two entrances). In the presented model, the reaction time of the load balancer is too slow to be beneficial at light loads, as it requires three accounting periods. When loads are equally distributed between the rooms, load balancing unnecessarily moves resources. The further calibration of a model to realistically reflect the domain is challenging. For example, it can be done in terms of observations of a reference implementation (see [12]).

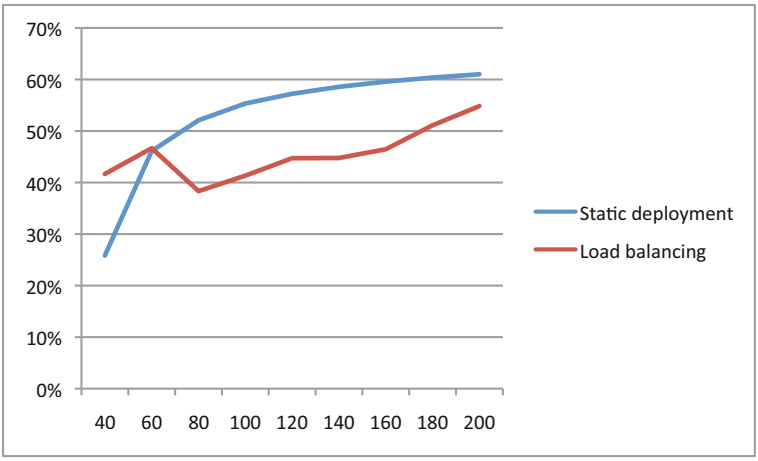


Fig. 10. Percentage of missed deadlines ranging over the number of avatars

6 Related Work

The concurrency model of ABS is akin to concurrent objects and Actor-based computation, in which software units with encapsulated processors communicate asynchronously [5, 18, 21, 32]. Their inherent compositionality allows concurrent objects to be naturally distributed on different locations, because only the local state of a concurrent object is needed to execute its methods. In previous work [4, 22, 23], the authors have introduced *deployment components* as a modeling concept for deployment architectures, which captures restricted resources shared between a group of concurrent objects, and shown how components with parametric resources may be used to capture a model's behavior for different assumptions about the available resources. The formal details of this approach are given in [23]. In previous work, the cost of execution was fixed in the language semantics. In this paper, we generalize that approach by proposing the specification of resource costs as part of the software development process. This is supported by letting default costs be overridden by annotations with user-defined cost expressed in terms of the local state and the input parameters to methods. This way, the cost of execution in the model may be adapted by the modeler to a specific cost scenario. This allows us to abstractly model the effect of deploying concurrent objects on deployment components with different amounts of allocated resources at an early stage in the software development process, before modeling the detailed control flow of the targeted system. In two larger case studies addressing resource management in the cloud [12, 26], the presented approach is compared to specialized simulation tools and to measurements on deployed code.

Techniques for prediction or analysis of non-functional properties are based on either *measurement* or *modeling*. Measurement-based approaches apply to

existing implementations, using dedicated profiling or tracing tools like JMeter or LoadRunner. Model-based approaches allow abstraction from specific system intricacies, but depend on parameters provided by domain experts [15]. A survey of model-based performance analysis techniques is given in [6]. Formal systems using process algebra, Petri Nets, game theory, and timed automata have been used in the embedded software domain (e.g., [9,16]), but also to the schedulability of processes in concurrent objects [19]. The latter work complements ours as it does not consider restrictions on shared deployment resources, but associates deadlines with method calls with abstract duration statements.

Work on modeling object-oriented systems with resource constraints is more scarce. Eckhardt et al. [14] use statistical modeling of meta-objects and virtual server replication to maintain service availability under denial of service attacks. Using the UML SPT profile for schedulability, performance, and time, Petriu and Woodside [29] informally define the Core Scenario Model (CSM) to solve questions that arise in performance model building. CSM has a notion of resource context, which reflects an operation's set of resources. CSM aims to bridge the gap between UML and techniques to generate performance models [6]. Closer to our work is M. Verhoef's extension of VDM++ for embedded real-time systems [31], in which static architectures are explicitly modeled using CPUs and buses. The approach uses fixed resources targeting the embedded domain, namely processor cycles bound to the CPUs, while we consider more general resources for arbitrary software. Verhoef's approach is also based on abstract executable modeling, but the underlying object models and operational semantics differ. VDM++ has multi-thread concurrency, preemptive scheduling, and a strict separation of synchronous method calls and asynchronous signals, in contrast to our work with concurrent objects, cooperative scheduling, and caller-decided synchronization.

Others interesting lines of research are static cost analysis (e.g., [3, 17]) and symbolic execution for object-oriented programs. Most tools for cost analysis only consider sequential programs, and assume that the program is fully developed before cost analysis can be applied. COSTABS [2] is a cost analysis tool for ABS which supports concurrent object-oriented programs. Our approach, in which the modeler specifies cost in annotations, can be combined with COSTABS to derive cost annotations for the parts of a model that are fully implemented. In collaboration with Albert *et al.*, this approach has been used for memory analysis of ABS models [4]. Extending our approach with symbolic execution would allow the best- and worst-case response time to be calculated for the different balancing strategies depending on the available resources and user load.

7 Conclusion

This paper gives an overview of a simple and flexible approach to integrating deployment architectures and resource consumption into executable object-oriented models. The approach is based on a separation of concerns between the *resource cost* of performing computations and the *resource capacity* of the

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

By integrating deployment architectures into software models, application-level resource management policies become an integral part of the software design. This concept is illustrated through the running example of the paper, a virtual world which is extended by a load balancing strategy based on group collaboration. The load balancing strategy is compared to static deployment by means of simulations which count the deadline misses in the model.

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

Acknowledgement. The contributions of Rudi Schlatte and Lizeth Tapia, who have been directly involved in the development of deployment modeling in ABS, are gratefully acknowledged. The author would further like to thank the HATS project members for a friendly and fruitful working environment. This work has in particular profited from collaboration with Peter Wong, Elvira Albert and the COSTA team, Reiner Hähnle, Frank de Boer, and Olaf Owe.

References

1. Agha, G.A.: ACTORS: A Model of Concurrent Computations in Distributed Systems. The MIT Press, Cambridge (1986)
2. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: COSTABS: A cost and termination analyzer for ABS. In: Kiselyov, O., Thompson, S. (eds.) Proc. Workshop on Partial Evaluation and Program Manipulation (PEPM 2012), pp. 151–154. ACM (2012)
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Java Bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007)
4. Albert, E., Genaim, S., Gómez-Zamalloa, M., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Simulating concurrent behaviors with worst-case cost bounds. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 353–368. Springer, Heidelberg (2011)
5. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (2007)

6. Balsamo, S., Marco, A.D., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: A survey. IEEE Transactions on Software Engineering 30(5), 295–310 (2004)
7. Bjørk, J., de Boer, F.S., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: User-defined schedulers for real-time concurrent objects. Innovations in Systems and Software Engineering 9(1), 29–43 (2013)
8. Caromel, D., Henrio, L.: A Theory of Distributed Object. Springer (2005)
9. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Resource interfaces. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 117–133. Springer, Heidelberg (2003)
10. Clarke, D., Diakov, N., Hähnle, R., Johnsen, E.B., Schaefer, I., Schäfer, J., Schlatte, R., Wong, P.Y.H.: Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 417–457. Springer, Heidelberg (2011)
11. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
12. de Boer, F.S., Hähnle, R., Johnsen, E.B., Schlatte, R., Wong, P.Y.H.: Formal modeling of resource management for cloud architectures: An industrial case study. In: De Paoli, F., Pimentel, E., Zavattaro, G. (eds.) ESOCC 2012. LNCS, vol. 7592, pp. 91–106. Springer, Heidelberg (2012)
13. de Boer, F.S., Jaghoori, M.M., Johnsen, E.B.: Dating concurrent objects: Real-time modeling and schedulability analysis. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 1–18. Springer, Heidelberg (2010)
14. Eckhardt, J., Mühlbauer, T., AlTurki, M., Meseguer, J., Wirsing, M.: Stable availability under denial of service attacks through formal patterns. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 78–93. Springer, Heidelberg (2012)
15. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by run-time parameter adaptation. In: Proc. 31st International Conference on Software Engineering (ICSE 2009), pp. 111–121. IEEE (2009)
16. Fersman, E., Krcál, P., Pettersson, P., Yi, W.: Task automata: Schedulability, decidability and undecidability. Information and Computation 205(8), 1149–1172 (2007)
17. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In: POPL, pp. 127–139. ACM (2009)
18. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theoretical Computer Science 410(2-3), 202–220 (2009)
19. Jaghoori, M.M., de Boer, F.S., Chothia, T., Sirjani, M.: Schedulability of asynchronous real-time concurrent objects. Journal of Logic and Algebraic Programming 78(5), 402–416 (2009)
20. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
21. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and Systems Modeling 6(1), 35–58 (2007)
22. Johnsen, E.B., Owe, O., Schlatte, R., Tapia Tarifa, S.L.: Dynamic resource reallocation between deployment components. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 646–661. Springer, Heidelberg (2010)

23. Johnsen, E.B., Owe, O., Schlatte, R., Tapia Tarifa, S.L.: Validating timed models of deployment components with parametric concurrency. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 46–60. Springer, Heidelberg (2011)
24. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: A formal model of object mobility in resource-restricted deployment scenarios. In: Arbab, F., Ölveczky, P.C. (eds.) FACS 2011. LNCS, vol. 7253, pp. 187–204. Springer, Heidelberg (2012)
25. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: A formal model of user-defined resources in resource-restricted deployment scenarios. In: Beckert, B., Damiani, F., Gurov, D. (eds.) FoVeOOS 2011. LNCS, vol. 7421, pp. 196–213. Springer, Heidelberg (2012)
26. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Modeling Resource-Aware Virtualized Applications for the Cloud in Real-Time ABS. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 71–86. Springer, Heidelberg (2012)
27. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Modeling application-level management of virtualized resources in ABS. In: Beckert, B., Bonsangue, M.M. (eds.) FMCO 2011. LNCS, vol. 7542, pp. 89–108. Springer, Heidelberg (2012)
28. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer 1(1-2), 134–152 (1997)
29. Petriu, D.B., Woodside, C.M.: An intermediate metamodel with scenarios and resources for generating performance models from UML designs. Software and System Modeling 6(2), 163–184 (2007)
30. Vellon, M., Marple, K., Mitchell, D., Drucker, S.: The architecture of a distributed virtual worlds system. In: Proc. 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS), pp. 211–218. USENIX (1998)
31. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and validating distributed embedded real-time systems with VDM++. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 147–162. Springer, Heidelberg (2006)
32. Welc, A., Jagannathan, S., Hosking, A.: Safe futures for Java. In: Proc. Object oriented programming, systems, languages, and applications (OOPSLA 2005), pp. 439–453. ACM Press, New York (2005)

Composing Distributed Systems: Overcoming the Interoperability Challenge

Valérie Issarny and Amel Bennaceur

Inria Paris-Rocquencourt, France
firstname.lastname@inria.fr

Abstract. Software systems are increasingly composed of independently-developed components, which are often systems by their own. This composition is possible only if the components are *interoperable*, i.e., are able to work together in order to achieve some user task(s). However, interoperability is often hampered by the differences in the data types, communication protocols, and middleware technologies used by the components involved. In order to enable components to interoperate despite these differences, mediators that perform the necessary data translations and coordinate the components' behaviours appropriately, have been introduced. Still, interoperability remains a critical challenge for today's and even more tomorrow's distributed systems that are highly heterogeneous and dynamic. This chapter introduces the fundamental principles and solutions underlaying interoperability in software systems with a special focus on protocols. First, we take a software architecture perspective and present the fundamentals for reasoning about interoperability and bring out mediators as a key solution to achieve protocol interoperability. Then, we review the solutions proposed for the implementation, synthesis, and dynamic deployment of mediators. We show how these solutions still fall short in automatically solving the interoperability problem in the context of systems of systems. This leads us to present the solution elaborated in the context of the European CONNECT project, which revolves around the notion of emergent middleware, whereby mediators are synthesised on the fly. We consider the GMES (Global Monitoring of Environment and Security) initiative and use it to illustrate the different solutions presented.

Keywords: Architectural mismatches, Interoperability, Mediator synthesis, Middleware.

1 Introduction

Modern software systems are increasingly composed of many components, which are distributed across the network and collaborate to perform a particular task. These components, often being complex systems themselves, led to the emergence of what is known as "systems of systems" [32]. The realisation of a system of systems depends on the ability to achieve *interoperability* between its different component systems. Traditionally, *"Interoperability characterises the extent by*

E. Giachino et al. (Eds.): FMCO 2012, LNCS 7866, pp. 168–196, 2013.

which two implementations of systems or components from different manufacturers can co-exist and work together by merely relying on each other's services as specified by a common standard" [48]. However, assuming the reliance on a common standard is often unrealistic given that components are increasingly highly heterogeneous (from the very tiny thing to the very large cloud) and developed (from design to deployment) independently without knowing the systems with which they will be composed. As a result, even though the services that component systems (or *components* for short) require or provide to each other are compatible at some high-level of abstraction, their implementations may encompass many differences that prevent them from working together. Hence, we re-state the definition of interoperability to *components that require and provide compatible high-level functionalities and can be made to work together*. We qualify such components as being *functionally compatible*.

In order to make functionally-compatible components work together, we must reconcile the differences between their implementations. The differences may be related to the data types or the format in which the data are encapsulated, that is, *data heterogeneity*. Differences may also concern the protocols according to which the components interact, that is, *behavioural heterogeneity*, which is the main focus of this chapter. Middleware, which is a software logically placed between the higher layer consisting of users and applications, and the layer underneath consisting of operating systems and basic communication facilities, provides an abstraction that facilitates the development of applications despite the heterogeneity of the underlying infrastructure. However, the abstractions defined by the middleware constrain the structure of the data that components exchange and the coordination paradigm according to which they communicate. This makes it impossible for components implemented using different middleware technologies to interoperate. As a result, interoperability must be considered at both application and middleware layers. To achieve interoperability between components featuring data and behavioural heterogeneity, intermediary software entities, called *mediators*, are used to perform the necessary translations of the data exchanged and to coordinate the components' protocols appropriately [52]. Using mediators to achieve interoperability has received a great deal of interest and led to the definition of a multitude of solutions, both theoretical and practical, for the specification, synthesis, and deployment of mediators, although they are predominantly oriented toward design time.

With the growing emphasis on spontaneous interaction whereby components are discovered at runtime and need to be composed dynamically, mediators can no longer be specified or implemented at design time. Rather, they have to be synthesised and deployed on the fly. Therefore, the knowledge necessary for the synthesis of mediators must be represented in a form that allows its automated processing. Research on knowledge representation in general, and ontologies in particular, has now made it possible to model and automatically reason about domain information crisply, if not with the same nuanced interpretation that a developer might [45]. Semantic Web Services are an example of the use of ontologies in enabling mediation on the fly [38]. However, they are restricted to

interoperability at the application layer, assuming that the composed components are implemented using the same middleware.

Acknowledging the extensive work on fostering interoperability, while at the same time recognising the increasing challenge that it poses to developers, this chapter provides a comprehensive review of the interoperability challenge, from its formal foundations to its automated support through the synthesis of mediators. The work that is reported extensively builds on the result of the European collaborative project CONNECT[1], which introduced the concept of *emergent middleware* and related enablers so as to sustain interoperability in the increasingly connected digital world. An emergent middleware is a dynamically generated distributed system infrastructure for the current operating environment and context, which allows functionally-compatible systems to interoperate seamlessly.

This chapter is organised as follows. In Section 2, we introduce the GMES case study, which we use throughout the chapter to illustrate the different solutions to interoperability. In Section 3, we take a software architecture perspective to understand and further formalise the interoperability problem in the case of systems of systems, which are characterised by the extreme heterogeneity of their components and the high-degree of dynamism of the operating environment. In Section 4, we survey the approaches to achieving interoperability from (i) a *middleware perspective* where we concentrate on the effort associated with the implementation of mediators, (ii) a *protocol perspective* where we are concerned with the synthesis of mediators based on the behavioural specification of component systems, thereby greatly facilitating the developer's task and further promoting software correctness, and (iii) a Semantic Web perspective where we focus on the fully automated synthesis of mediators at runtime, so as to enable on-the-fly composition of component systems in the increasingly open and dynamic networking environment. Following this state of the art review, in Section 5, we introduce a multifaceted approach to interoperability which brings together the different perspectives in order to provide a solution to interoperability based on the automated synthesis of mediators and their dynamic deployment, which we call emergent middleware. Finally, in Section 6, we conclude on where interoperability stands in today's systems and present directions for future work.

2 GMES: A System of Systems Case Study

To highlight the interoperability challenge in systems of systems, we consider one representative application domain, that of global monitoring of the natural environment, as illustrated by the GMES[2] initiative. GMES is the European Programme for the establishment of a European capacity for Earth Observation. A special interest is given to the support of emergency situations across different European countries [22]. In emergency situations, the context is highly dynamic and involves highly heterogeneous components that interact in order to perform the different tasks necessary for decision making. The tasks include,

[1] `http://www.connect-forever.eu/`

[2] Global Monitoring for Environment and Security – `http://www.gmes.info/`

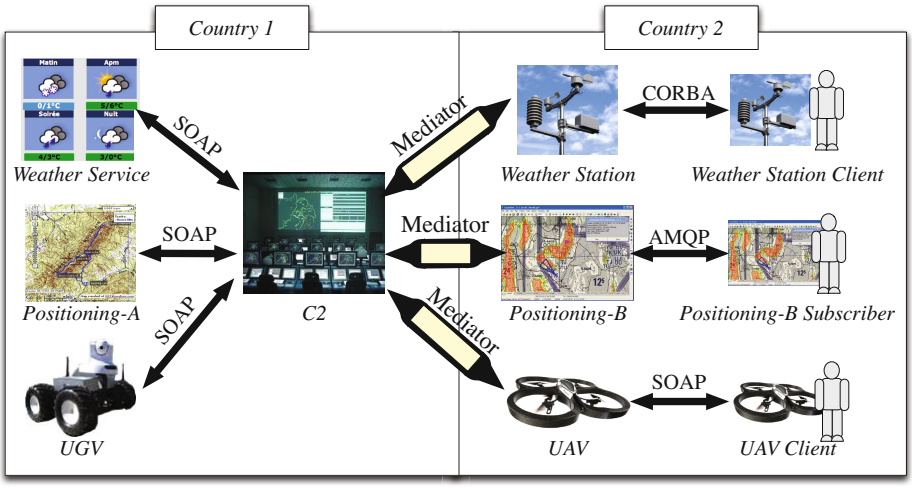


Fig. 1. The GMES use case

among others, collecting weather information, and capturing video using different devices. GMES makes a strong case of the need for on-the-fly solutions to interoperability in systems of systems. Indeed, each country defines an emergency management system that composes different components, which interact according to standards proper to the country. However, in special circumstances, assistance may come from other countries, which bring their own components defined using different standards.

Figure 1 depicts the case where the emergency system of *Country 1* is composed of a Command and Control centre (*C2*) which takes the necessary decisions for managing the crisis based on the information about the weather provided by the *Weather Service* component, the positions of the various agents in field given by *Positioning-A*, and the video of the operating environment captured by *UGV* (Unmanned Ground Vehicle) robots with sensing capabilities. The different components use SOAP[3] to communicate. *Country 2* assists *Country 1* by supplying components that provide the *C2* component with extra information. These components consists in *Weather Station*, the *Positioning-B* positioning system, and a *UAV* (Unmanned Aerial Vehicle) drone. However, *C2* cannot use these components directly [23]. Indeed, *Weather Station* that is implemented using CORBA[4], provides specific information such as temperature or humidity whereas *Weather Service*, which is used by *C2*, returns all of this information using a single operation. Further, *Positioning-A* is implemented using SOAP and interacts according to the request/response paradigm whereas *Positioning-B* is implemented using AMQP[5] and hence interacts according to the publish/subscribe paradigm. Also,

[3] `http://www.w3.org/TR/soap/`
[4] `http://www.omg.org/technology/documents/corba_spec_catalog.htm`
[5] `http://www.amqp.org`

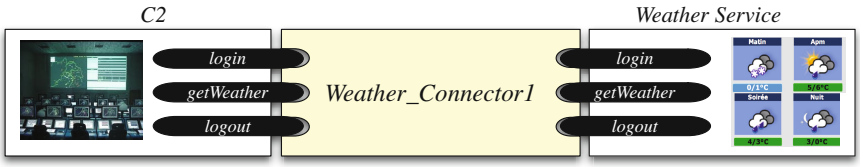


Fig. 2. *C2* and *Weather Service*, and associated connector in *Country 1*

UGV requires the client to login, then it can move in the four cardinal directions while *UAV* is required to takeoff prior to any operation and to land before logging out. To enable *C2* to use the components provided by *Country 2*, with which it is functionally compatible, mediators have to be synthesised and deployed in order to make *C2* interoperate with *Weather Station*, *Positioning-B*, and *UAV*.

3 The Interoperability Problem Space: A Software Architecture Perspective

Software systems may be abstractly described at the architectural level in terms of components and connectors: components are meant to encapsulate computation while connectors are meant to encapsulate interaction. In other words, control originates in components, and connectors are channels for coordinating the control flow (as well as data flow) between components [46].

So as to sustain software composition, software components must not only specify their (provided) interfaces, i.e., the subset of the system's functionality and/or data that is made accessible to the environment, but also the assumptions they make in terms of other components/interfaces that must be present for a component to fulfill its functionality, making all dependencies explicit [49,24]. A software connector is an architectural element tasked with effecting and regulating interactions among components via their interfaces [49]. As an illustration, Figure 2 depicts the case of the *C2* component interacting with *Weather Service*, where *C2* exhibits an interface in which it requires the operations *login*, *getWeather*, and *logout*, and *Weather Service* defines an interface in which it provides these same operations. The connector *Weather_Connector1* coordinates these operations based on the SOAP middleware technology.

From the standpoint of implementation, middleware provides the adequate basis for implementing connectors. Indeed, middleware greatly eases the composition of components by introducing abstractions that hide the heterogeneity of the underlying infrastructure. However, middleware defines specific data formats and interaction paradigms, making it difficult for components developed using different middleware to communicate [42].

In general, the assembly of components via connectors may conveniently be reasoned about based on the appropriate formalisation of software architecture, as discussed in the following section.

Table 1. FSP syntax overview

Definitions	
αP	The alphabet of a process P
END	Predefined process. Denotes the state in which a process successfully terminates
set S	Defines a set of action labels
$[i : S]$	Binds the variable i to a value from S
Primitive Processes (P)	
$a \rightarrow P$	Action prefix
$a \rightarrow P \vert b \rightarrow P$	Choice
$P; Q$	Sequential composition
$P(X =' a)$	Parameterised process: P is described using parameter X and modelled for a particular parameter value, $P(a1)$
$P/\{new\_1/old\_1, ..., new\_n/old\_n\}$	Relabelling
$P \backslash \{a_1, a_2, ..., a_n\}$	Hiding
$P + \{a_1, a_2, ..., a_n\}$	Alphabet extension
Composite Processes ($\|P$)	
$P \| Q$	Parallel composition
forall $[i : 1..n]$ $P(i)$	Replicator construct: equivalent to the parallel composition $(P(1) \| ... \| P(n))$.
$a : P$	Process labelling

3.1 Formal Foundations for Software Architectures

To enable formal reasoning about software architecture composition, the interaction protocols implemented by components and connectors may be specified using a process algebra, as introduced in the pioneering work of Allen and Garlan [1]. In the context of this chapter, we concentrate more specifically on the use of FSP (Finite State Processes) based on the work of Spitznagel and Garlan, which in particular considers the adaptation of connectors to address dependability as well as interoperability concerns [47].

Finite State Processes. FSP [35] is a process algebra that has proven to be a convenient formalism for specifying concurrent components, analysing, and reasoning about their behaviours. Table 1 provides an overview of the FSP operators, while the interested reader is referred to [35] for further detail. Briefly stated, FSP processes describe actions (events) that occur in sequence, and choices between action sequences. Each process has an alphabet, αP, of the actions that it is aware of (and either engages in or refuses to engage in). There are two types of processes: *primitive processes* and *composite processes*. Primitive processes are constructed through action prefix, choice, and sequential composition. Composite processes are constructed using parallel composition or process relabelling. The replicator forall

is a convenient syntactic construct used to specify parallel composition over a set of processes. Processes can optionally be parameterised and have re-labelling, hiding or extension over their alphabet. A composite process is distinguished from a primitive process by prefixing its definition with ‖.

The semantics of FSP is given in terms of Labelled Transition Systems (LTS) [33]. The LTS interpreting an FSP process P can be regarded as a directed graph whose nodes represent the process states and each edge is labelled with an action $a \in \alpha P$ representing the behaviour of P after it engages in a. $P \xrightarrow{a} P'$ then denotes that P transits with action a into P'. Then, $P \overset{s}{\Rightarrow} P'$ is a shorthand for $P \xrightarrow{a_1} P_1 \xrightarrow{a_2} P_2 ... \xrightarrow{a_n} P', s = \langle a_1, a_2, ..., a_n \rangle, a_i \in \alpha P$. There exists a start node from which the process begins its execution. The END state indicates a successful termination. When composed in parallel, processes synchronise on shared actions: if processes P and Q are composed in parallel, actions that are in the alphabet of only one of the two processes can occur independently of the other process, but an action that is in the alphabets of both processes cannot occur until the two of them are willing to engage in it, as described below:

$$\frac{P \xrightarrow{a} P', \nexists a \in \alpha Q}{P \| Q \xrightarrow{a} P' \| Q} \qquad \frac{Q \xrightarrow{a} Q', \nexists a \in \alpha P}{P \| Q \xrightarrow{a} P \| Q'} \qquad \frac{P \xrightarrow{a} P', Q \xrightarrow{a} Q'}{P \| Q \xrightarrow{a} P' \| Q'}$$

Formalising Components and Connectors Using FSP. The interaction protocols run by components are described using a set of FSP processes called *ports*. For example, consider the port of *C2* dedicated to the interaction with *Weather Service* (see Figure 2): *C2* logs in, invokes the operation *getWeather* several times, and finally logs out. The port of *C2* is specified, using FSP, as follows:

$$\begin{aligned} C2\_port &= (req.login \rightarrow P1), \\ P1 \quad &= (req.getWeather \rightarrow P1 | req.logout \rightarrow C2\_port). \end{aligned}$$

We further use FSP processes to describe a connector as a set of *roles* and a *glue*. Roles are the processes that specify the expected local behaviours of the various interacting parties coordinated by the connector, while the glue process describes the specific coordination protocol that is implemented [1]. Still considering our example of Figure 2, the *Weather_Connector1* connector managing the interactions between *C2* and *Weather Service* defines a role associated with each of them, that is, *C2_role* and *WeatherService_role*, respectively. The connector also defines how these operations are realised using a SOAP request/response paradigm. More specifically, each required operation corresponds to the sending of a SOAP request parameterised with the name of the operation, and the reception of the corresponding SOAP response, which is specified by the process *SOAPClient*. The dual provided operation corresponds to the receiving of a SOAP request parameterised with the name of the operation, and the send of the corresponding SOAP response, which is specified by the process *SOAPServer*. Furthermore, a request sent from one side is received from the other and similarly for a response, which is specified by the process *SOAPGlue*. *Weather_Connector1* is then specified as the parallel composition of all these processes:

$$
\begin{array}{ll}
\textsf{set}\ weather\_actions1 & = \{login, getWeather, logout\} \\
C2\_role & = (req.login \rightarrow P1), \\
P1 & = (req.getWeather \rightarrow P1 | req.logout \rightarrow C2\_Role).
\end{array}
$$

$$
\begin{array}{ll}
WeatherService\_role & = (prov.login \rightarrow P2), \\
P2 & = (\ prov.getWeather \rightarrow P2 | prov.logout \rightarrow WeatherService\_role).
\end{array}
$$

$$
\begin{array}{ll}
SOAPClient\ (X =' op) & = (req.[X] \rightarrow sendSOAPRequest[X] \rightarrow receiveSOAPResponse[X] \\
 & \quad \rightarrow SOAPClient). \\
SOAPServer\ (X =' op) & = (prov.[X] \rightarrow receiveSOAPRequest[X] \rightarrow sendSOAPResponse[X] \\
 & \quad \rightarrow SOAPServer). \\
SOAPGlue\ (X =' op) & = (sendSOAPRequest[X] \rightarrow receiveSOAPRequest[X] \rightarrow \\
 & \quad sendSOAPResponse[X] \rightarrow receiveSOAPResponse[X] \rightarrow SOAPGlue).
\end{array}
$$

$$
\begin{array}{ll}
\|\, Weather\_Connector1 & = (\ C2\_role \\
 & \quad \| \quad WeatherService\_role \\
 & \quad \| \quad (\textsf{forall}[op : weather\_actions1] SOAPClient(\text{op})) \\
 & \quad \| \quad (\textsf{forall}[op : weather\_actions1] SOAPGlue(\text{op})) \\
 & \quad \| \quad (\textsf{forall}[op : weather\_actions1] SOAPServer(\text{op}))).
\end{array}
$$

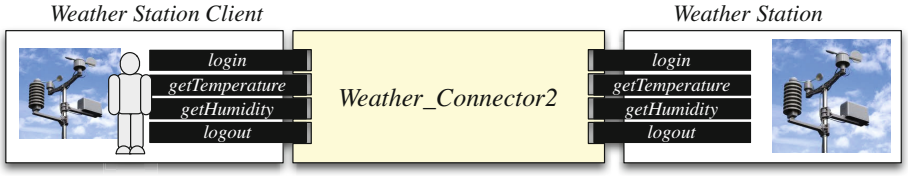


Fig. 3. The *Weather Station* and associated client in *Country 2*

Consider now the case of the *Weather Station* component interacting with its specific client (see Figure 1). As depicted in Figure 3, *Weather Station* exhibits an interface through which it provides the operations *login*, *getTemperature*, *getHumidity*, and *logout*, while the associated port is specified as follows:

$$
\begin{array}{ll}
WeatherStation\_port & = (prov.login \rightarrow P2), \\
P2 & = (\ prov.getTemperature \rightarrow P2 \\
 & \quad |\ \ prov.getHumidity \rightarrow P2 \\
 & \quad |\ \ prov.logout \rightarrow WeatherStation\_port).
\end{array}
$$

The connector *Weather_Connector2* then coordinates the operations between *Weather Station* and the corresponding client according to the CORBA request/response paradigm and is specified as follows:

$$
\begin{array}{ll}
\textsf{set}\ weather\_actions2 & = \{login, getTemperature, getHumidity, logout\} \\
WeatherStationClient\_role & = (req.login \rightarrow P1), \\
P1 & = (\ req.getTemperature \rightarrow P1 \\
 & \quad |\ \ req.getHumidity \rightarrow P1 \\
 & \quad |\ \ req.logout \rightarrow WeatherStationClient\_role). \\
WeatherStation\_role & = (prov.login \rightarrow P2), \\
P2 & = (\ prov.getTemperature \rightarrow P2 \\
 & \quad |\ \ prov.getHumidity \rightarrow P2 \\
 & \quad |\ \ prov.logout \rightarrow WeatherStation\_role).
\end{array}
$$

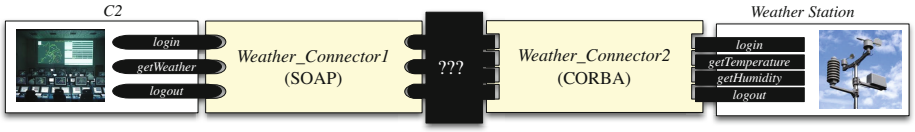


Fig. 4. Architectural mismatch between *C2* and *Weather Station*

$$
\begin{aligned}
CORBAClient\ (X =' op) &= (req.[X] \rightarrow sendCORBARequest[X] \\
&\quad \rightarrow receiveCORBAResponse[X] \rightarrow CORBAClient). \\
CORBAServer\ (X =' op) &= (prov.[X] \rightarrow receiveCORBARequest[X] \\
&\quad \rightarrow sendCORBAResponse[X] \rightarrow CORBAServer). \\
CORBAGlue\ (X =' op) &= (sendCORBARequest[X] \rightarrow receiveCORBARequest[X] \\
&\quad \rightarrow sendCORBAResponse[X] \rightarrow receiveCORBAResponse[X] \\
&\quad \rightarrow CORBAGlue). \\
\| Weather\_Connector2 &= (\ WeatherStationClient\_role \\
&\quad \| \quad WeatherStation\_role \\
&\quad \| \quad (\mathsf{forall}[op : weather\_actions2]\, CORBAClient(\mathrm{op})) \\
&\quad \| \quad (\mathsf{forall}[op : weather\_actions2]\, CORBAGlue(\mathrm{op})) \\
&\quad \| \quad (\mathsf{forall}[op : weather\_actions2]\, CORBAServer(\mathrm{op}))).
\end{aligned}
$$

Thanks to the formal specification of architectural components and connectors, architectural mismatches may be reasoned about. Specifically, architectural mismatches occur when composing two, or more, software components to form a system and those components make different assumptions about their environment [27], thereby preventing interoperability. These assumptions relate to: (i) the data and control models of the involved components, (ii) the protocols and the data model specified by the connector, and (iii) the infrastructure and the development environment on top of which the components are built. Consider for example the composition of *C2* and *Weather Station*. There exists an architectural mismatch between the two, which hampers their interoperation (see Figure 4). Indeed, the components manipulate different data: *C2* deals with weather whereas *Weather Station* manages temperature and humidity. They are also implemented using different middleware technologies: SOAP for *C2* and CORBA for *Weather Station*. In the following section, we show how to reason about the different assumptions that components make about their connection.

3.2 Reasoning about Architectural Mismatches

Architectural mismatches can be reasoned about formally by comparing component port and connector roles [1]. More specifically, a component can be attached to a connector only if its port is *behaviourally compatible* with the connector role it is bound to. Behavioural compatibility between a component port and a connector role is based upon the notion of refinement, i.e., a component port is behaviourally compatible with a connector role if the process specifying the behaviour of the former refines the process characterising the latter [1]. In other words, it should be possible to substitute the role process by the port process.

For example, the *C2* component can be attached to *Weather_Connector1* connector since *C2_port* refines *C2_role* — they are actually the same. Likewise, *WeatherStation_port* refines *WeatherStation_role* defined by

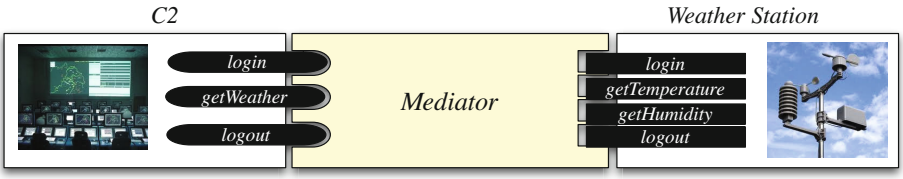


Fig. 5. Mediator to solve architectural mismatch between *C2* and *Weather Station*

Weather_Connector2. However, *WeatherStation_port* cannot be attached to *Weather_Connector1* since it does not refine any of its ports, nor *C2_role* can be attached to *Weather_Connector2*. Hence, in the case of *C2* willing to interact with *Weather Station*, none of the available connectors can readily be used resulting in an architectural mismatch, which needs to be overcome by a mediator, as depicted in Figure 5.

3.3 Mediators Adapting Connectors for Interoperability

In order to solve architectural mismatches without modifying the components themselves, it is necessary to construct a connector that reconciles the assumptions that each of the components makes about its environment. The connector need not necessarily be constructed from scratch. It can also be developed by transforming existing connectors. Hence, a connector with n roles coordinated using a *Glue* process and specified as: $Connector = R_1 \| ... \| R_n \| Glue$ can be adapted into a mediator: $Mediator = f_1(R_1) \| ... \| f_n(R_n) \| R_{n+1} \| ... \| R_{n+k} \| f_G(Glue)$ with which the ports of the components at hand are behaviourally compatible. For example, the mediator between *C2* and *Weather Station* includes roles behaviourally compatible with *C2_port* and *WeatherStation_port* and encompasses the glue that coordinates them.

Spitznagel and Garlan [47] introduce a set of transformation patterns (e.g., data translation and event aggregation), which a developer can specify and compose in order to construct complex connectors based on existing ones. The complex connectors that are specifically considered in [47] enhance the coordination of components with respect to enforcing stronger dependability guarantees. However, complex connectors may as well be built to overcome architectural mismatches. Still, the question that raises itself is which transformations (i.e., composition of the given connector with transformation patterns) are valid and which do not make sense for a specific mismatch at hand. For example, the mediator between *C2* and *Weather Station* has to translate the $getWeather$ operation required by *C2* into the $getTemperature$ and $getHumidity$ operations provided by *Weather Station*. Hence, it needs to compose the connectors *Weather_Connector1* and *Weather_Connector2* , respectively associated with *C2* and the *Weather Station*, with the process:

$$Map = (req.getWeather \rightarrow prov.getTemperature \rightarrow prov.getHumidity).$$

However, this only solves mismatches occurring at the application layer and mediation is also necessary at the middleware layer so as to bridge SOAP and CORBA. Another concern for the composition of connectors is the increasing dynamics of the networking environment, which calls for on-the-fly mediation.

3.4 Dynamic Software Architecture and Mediation

In dynamic environments where components are discovered at runtime and composed dynamically, mediators can no longer be specified or implemented at design time. Instead, they have to be synthesised and deployed on the fly. However, the synthesis of mediators not only requires knowledge of the data and behaviour of the components but also knowledge of the domain, which specifies the relation between the data and operations of the different components. In particular, *ontologies* build upon sound logical theory to provide a machine-interpretable means to reason, automatically, about the semantics of data based on the shared understanding of the domain [4]. As a matter of fact, ontologies prove valuable when dealing with data interoperability. In this context, ontologies are used to specify a shared vocabulary precisely and offer a common basis to reconcile data syntactic differences based on their semantic definitions. They further play a valuable role in software engineering by supporting the automated integration of knowledge among teams and project stakeholders [17]. For example, a weather ontology would allow us to infer the relation between $getWeather$ and $getTemperature$ and $getHumidity$ without a need for human intervention.

Ontologies have also been widely used for the modelling of Semantic Web Services and to achieve efficient service discovery and composition [41,40]. Semantic Web Services use ontologies as a central point to achieve interoperability between heterogeneous clients and services at runtime. For example, WSMO (Web Service Modelling Ontology) relies on ontologies to support runtime mediation based on pre-defined patterns. However, the proposed approach does not ensure that such mediation does not lead to an erroneous execution (e.g., deadlock) [20]. It further assumes that components are implemented using the same middleware, SOAP.

Overall, the issue of overcoming architectural mismatches to make interoperable components that are functionally compatible, is a cross-cutting concern where protocol mismatches need to be addressed at both application and middleware layers. Interoperability solutions must consider conjointly application and middleware layers: (i) the application layer provides the appropriate level of abstraction to reason about interoperability and automate the generation of mediators; and (ii) the middleware layer offers the necessary services for realising the mediation by selecting and instantiating the specific data structures and protocols. In addition, mediators need to be synthesised on the fly, as the networking environment is now open and dynamic, thereby leading to the assembly of component systems that are known to one another other at runtime, as opposed to design time. As discussed next, supporting such a dynamic cross-layer mediation requires a multifaceted solution.

4 The Interoperability Solution Space: A Multifaceted Review

Sustaining interoperability has received a great deal of attention since the emergence of distributed systems and further promotion of component-based and service-oriented software engineering. We may classify solutions to protocol interoperability according to three broad perspectives: (i) the middleware perspective is specifically concerned with the implementation of middleware-layer mediators, based on the introduction of frameworks that allow bridging components that are implemented on top of heterogeneous infrastructures, (ii) the protocol perspective is focused on the systematic synthesis of mediators based on the specification of the protocols implemented by components to be made interoperable, and (iii) the semantic perspective is oriented toward automated reasoning about the matchmaking of components, both functionally and behaviourally.

4.1 The Middleware Perspective: Implementing Protocol Mediators

By definition, middleware defines an infrastructure mediator that overcomes the heterogeneity occurring in the lower layer. While original middleware solutions primarily targeted data heterogeneity, later middleware solutions had to deal with behavioural heterogeneity due to the composition of component systems relying on heterogeneous middleware protocols, as exemplified by the GMES case study. We then identify two basic approaches for the implementation of protocol mediators: (i) *pairwise mediation* where a specific bridge is implemented for each pair of heterogeneous protocols that need to be composed, and (ii) *mediation through a reference protocol* where protocol interoperability is achieved by bridging any protocol that needs to be composed with a reference protocol. The former leads to highly customised mediators while the latter significantly decreases the development effort associated with mediation. In the following, we describe both approaches in more detail.

Pairwise Mediation. Under pairwise mediation, the developer has to define the transformations necessary to reconcile the data and behaviour of the protocols involved and to ensure the correctness of these transformations. Mediation must be addressed at all the layers of protocol heterogeneity. This especially stands for the application and middleware layers, while lower network layer heterogeneity remains largely addressed through IP-based networking. For example, at the middleware layer, OrbixCOMet[6] performs the necessary translation between DCOM and CORBA and SOAP2CORBA[7] ensures interoperability between SOAP and CORBA in both directions.

Figure 6 depicts the example of the composition of *C2* with *Weather Station* using SOAP2CORBA, which allows the SOAP requests issued by *C2* to be translated into CORBA requests, and the corresponding CORBA responses to

[6] `http://www.iona.com/support/whitepapers/ocomet-wp.pdf`
[7] `http://soap2corba.sourceforge.net/`

Fig. 6. Pairwise mediation between layered protocols

be translated into SOAP responses. This translation is relative to a specific operation. Hence, this translation can be specified as follows:

$$SOAP2CORBA(X =' op) = (receiveSOAPRequest[X] \rightarrow sendCORBARequest[X] \rightarrow receiveCORBAResponse[X] \rightarrow sendSOAPResponse[X] \rightarrow SOAP2CORBA).$$

However, the connector *Weather_Connector12_Pairwise* defined as:

$$\begin{aligned} \| Weather\_Connector12\_Pairwise = (\ & C2\_role \\ \| \ & WeatherStation\_role \\ \| \ & (\mathsf{forall}[op : weather\_actions1]SOAPClient(\mathrm{op})) \\ \| \ & (\mathsf{forall}[op : weather\_actions1]SOAPGlue(\mathrm{op})) \\ \| \ & (\mathsf{forall}[op : weather\_actions1]SOAPServer(\mathrm{op}))). \\ \| \ & (\mathsf{forall}[op : weather\_actions2]CORBAClient(\mathrm{op})) \\ \| \ & (\mathsf{forall}[op : weather\_actions2]CORBAGlue(\mathrm{op})) \\ \| \ & (\mathsf{forall}[op : weather\_actions2]CORBAServer(\mathrm{op}))). \\ \| \ & SOAP2CORBA \end{aligned}$$

is not a valid connector since the translation is carried out assuming that the components refer to the same application-layer operations to coordinate. For example, when *C2* sends a SOAP request for *getWeather*, it is translated into a CORBA request for *getWeather*, but there is no counter part on the server side since *Weather Station* does not provide this operation. Indeed, higher application-layer mediation also needs to be implemented.

In general, the implementation of pairwise mediators is a complex task: developers have to deal with a lot of details and therefore must have a thorough understanding of the protocols at hand. As a result, solutions that help developers defining middleware-layer mediators have emerged. These solutions consist in a framework whereby the developer provides a declarative specification of the message translation across protocols, based on which the actual transformations are computed. For example, z2z [16] introduces a domain-specific language to describe the protocols to be made interoperable as well as the translation logic to compose them and then generates the corresponding bridge. Starlink [14] uses the same domain-specific models to specify bridges, which it deploys dynamically and interprets at runtime (see Figure 7). However, these solutions still require the developer to specify the translations to be made and hence to know both middleware in advance.

Mediation through a Reference Protocol. To reduce the development effort induced by pairwise mediation, a reference protocol can be used as an intermediary to translate from one protocol to another. Such mediation is especially appropriate for the middleware layer where heterogeneous middleware protocols

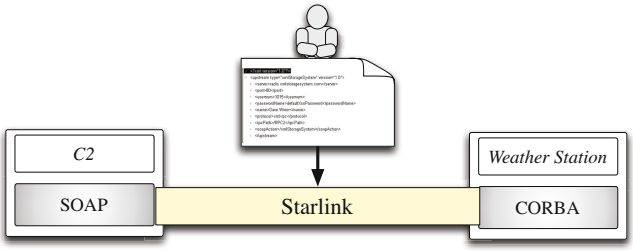


Fig. 7. Pairwise middleware-layer mediation based on high-level specification

may rather easily be mapped onto a common protocol when the middleware implement the same interaction paradigm. Application-layer reference protocols may also be considered for commonly encountered applications like messaging systems [7].

Enterprise Service Buses (ESBs), e.g., Oracle Service Bus[8] and IBM WebSphere Enterprise Service Bus[9], represent the most mature and common use of mediation through a reference protocol. An ESB [39] is an open standard, message-based, distributed integration infrastructure that provides routing, invocation and mediation services to facilitate the interactions of disparate distributed applications and services.

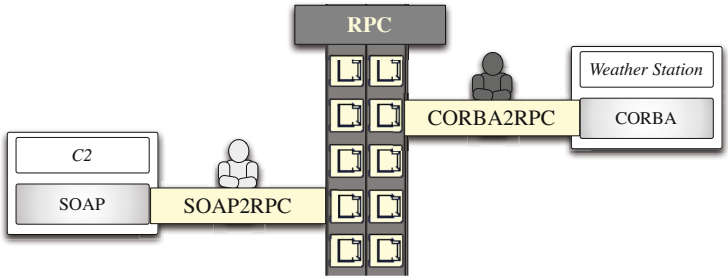


Fig. 8. Mediation through a reference protocol

When the intermediary reference protocol is defined independently of the set of middleware for which it guarantees interoperability, it does not necessarily capture all their details and specificities. Bromberg [12] puts forward the inference of the best-suited intermediary protocol based on the behaviours of the middleware involved. The author applies this approach to ensure interoperability in pervasive systems between different service discovery protocols using INDISS [15] and across RPC protocols (assuming the same application atop) using NEMESYS [12].

[8] `http://www.oracle.com/technetwork/middleware/service-bus/`
[9] `http://www-01.ibm.com/software/integration/wsesb/`

Going back to our example of the *C2* component willing to interact with *Weather Station*, both of them interact according to the RPC paradigm, which we can use as a reference protocol for ensuring interoperability between SOAP and CORBA (see Figure 8). Hence, we define the RPC (reference) connector as follows:

$$
\begin{array}{lll}
\mathsf{set}\ interface & = & \{any\} \\
Client\ (X =' op) & = & (sendRequest[X] \rightarrow receiveResponse[X] \rightarrow Client). \\
Server\ (X =' op) & = & (receiveRequest[X] \rightarrow sendResponse[X] \rightarrow Server). \\
Glue\ (X =' op) & = & (sendRequest[X] \rightarrow receiveRequest[X] \\
 & & \rightarrow sendResponse[X] \rightarrow receiveResponse[X] \rightarrow Glue). \\
\|RPCConnector & = & (\ (\mathsf{forall}[op : interface]Client(\mathrm{op})) \\
 & & \|\ \ (\mathsf{forall}[op : interface]Glue(\mathrm{op})) \\
 & & \|\ \ (\mathsf{forall}[op : interface]Server(\mathrm{op}))).
\end{array}
$$

We also need to define the transformations between each protocol and the reference protocol as follows:

$$
\begin{array}{lll}
SOAP2RPC(X =' op) & = & (sendSOAPRequest[X] \rightarrow translateSOAP2Request \\
 & & \rightarrow sendRequest[X] \rightarrow SOAP2RPC \\
 & & |\ receiveRequest[X] \rightarrow translate2SOAPRequest \\
 & & \rightarrow receiveSOAPRequest[X] \rightarrow SOAP2RPC \\
 & & |\ sendSOAPResponse[X] \rightarrow translateSOAP2Response \\
 & & \rightarrow sendResponse[X] \rightarrow SOAP2RPC \\
 & & |\ receiveSOAPResponse[X] \rightarrow translate2SOAPResponse \\
 & & \rightarrow receiveResponse[X] \rightarrow SOAP2RPC). \\
 & & \\
CORBA2RPC(X =' op) & = & (sendCORBARequest[X] \rightarrow translateCORBA2Request \\
 & & \rightarrow sendRequest[X] \rightarrow CORBA2RPC \\
 & & |\ receiveRequest[X] \rightarrow translate2CORBARequest \\
 & & \rightarrow receiveCORBARequest[X] \rightarrow CORBA2RPC \\
 & & |\ sendCORBAResponse[X] \rightarrow translateCORBA2Response \\
 & & \rightarrow sendResponse[X] \rightarrow CORBA2RPC \\
 & & |\ receiveCORBAResponse[X] \rightarrow translate2CORBAResponse \\
 & & \rightarrow receiveResponse[X] \rightarrow CORBA2RPC).
\end{array}
$$

We obtain the *Weather_Connector12_Reference* connector:

$$
\begin{array}{lll}
\|Weather\_Connector12\_Reference & = & (\ C2\_role \\
 & & \|\ \ Weather Station\_role \\
 & & \|\ \ (\mathsf{forall}[op : weather\_actions1]SOAPClient(\mathrm{op})) \\
 & & \|\ \ (\mathsf{forall}[op : weather\_actions1]SOAP2RPC(\mathrm{op})) \\
 & & \|\ \ (\mathsf{forall}[op : weather\_actions2]CORBAServer(\mathrm{op})) \\
 & & \|\ \ (\mathsf{forall}[op : weather\_actions2]CORBA2RPC(\mathrm{op})) \\
 & & \|\ \ (\mathsf{forall}[op : weather\_actions1]Client(\mathrm{op})) \\
 & & \|\ \ (\mathsf{forall}[op : weather\_actions1]Glue(\mathrm{op})) \\
 & & \|\ \ (\mathsf{forall}[op : weather\_actions1]Server(\mathrm{op})) \\
 & & \|\ \ (\mathsf{forall}[op : weather\_actions2]Client(\mathrm{op})) \\
 & & \|\ \ (\mathsf{forall}[op : weather\_actions2]Glue(\mathrm{op})) \\
 & & \|\ \ (\mathsf{forall}[op : weather\_actions2]Server(\mathrm{op}))).
\end{array}
$$

However, as in the case of *Weather_Connector12_Pairwise*, the *Weather_Connector12_Reference* connector only solves interoperability at the middleware layer and must be further enhanced to deal with interoperability at the application layer.

To sum up, a great amount of work exists on the development of concrete interoperability solutions to overcome middleware heterogeneity [9]. All these approaches tackle middleware interoperability assuming the use of the same application on top, while for components to be able to work together, differences

at both application and middleware layers need to be addressed. Similar approaches may be applied for application-layer protocols and actually are, but this is restricted to specific applications that are commonly encountered nowadays, like messaging applications [7]. In general, interoperability solutions based on the implementation of mediators do not scale to the unbounded universe of applications. Another issue is that middleware heterogeneity is often tackled for middleware defining the same interaction paradigm, while systems envisioned for the Future Internet are increasingly heterogeneous and require to compose systems based on distinct paradigms. The notion of extensible service buses enabling highly heterogeneous systems to interoperate across interaction paradigms has recently emerged but it is in its infancy [28]. The provision of interoperability solutions remains, however, a complex task for which automated support is of a great help.

4.2 The Protocol Perspective: Synthesising Protocol Mediators

In order to ease the task of the developers in achieving interoperability between functionally-compatible components, one approach is to provide methods and tools for the automated synthesis of mediators based on the specification of the protocols involved. The approaches can be applied at the application and middleware layers as long as they are isolated.

Lam [34] defines an approach for the synthesis of mediators using a reference protocol, which represents the glue of the mediator. Developers define the reference protocol based on an intuitive understanding of the features common to the protocols at hand. The author defines an approach for computing the relabelling function that maps the individual protocol onto the reference protocol. The mediator is then composed of the relabelling functions together with the reference protocol. As discussed in Section 4.1, at the middleware layer, we can specify the following reference protocol, which represents the glue of an RPC protocol:

$$\begin{aligned} \|\mathit{Reference\_Middleware\_protocol}\ (X =' op) = (&\mathit{sendRequest}[X] \rightarrow \mathit{receiveRequest}[X] \\ &\rightarrow \mathit{sendResponse}[X] \rightarrow \mathit{receiveResponse}[X] \\ &\rightarrow \mathit{Reference\_Mdw\_protocol}). \end{aligned}$$

However at the application layer, the synthesis cannot be applied as relabelling involves the translation of one operation only and not a sequence of operations. That is *getWeather*, *getTemperature*, and *getHumidity* cannot be mapped to the same operation. Hence, the *Weather_Connector12_Synthesised* connector does not allow *C2* and *WeatherStation* to interoperate:

$$\begin{aligned} \|\mathit{Weather\_Connector12\_Synthesised} = (\ & \mathit{C2\_role} \\ \|\ & \mathit{WeatherStation\_role} \\ \|\ & (\mathsf{forall}[op : \mathit{weather\_actions1}]\mathit{SOAPClient}(\mathrm{op})) \\ & /\{\mathit{sendSOAPRequest}/\mathit{sendRequest}, \\ \|\ & \ \ \mathit{receiveSOAPResponse}/\mathit{receiveResponse}\} \\ \|\ & (\mathsf{forall}[op : \mathit{weather\_actions2}]\mathit{CORBAServer}(\mathrm{op})) \\ & /\{\mathit{receiveCORBARequest}/\mathit{receiveRequest}, \\ & \ \ \mathit{sendCORBAResponse}/\mathit{sendResponse}\} \\ \|\ & (\mathsf{forall}[op : \mathit{weather\_actions1}]\mathit{Glue}(\mathrm{op})) \end{aligned}$$

The aforementioned solution consists in defining an abstraction common to the protocols to be mediated. An alternative approach is to synthesise a pairwise mediator based on a partial specification of the translations to be made, either as a goal, which represents a global specification of the composed system, or as an interface mapping, which represents the correspondence between the operations of the components. In the case of interoperability between *C2* and *Weather Station*, consider for example the following goal:

$$
\begin{array}{l}
Req1 = (Country1.req.login \rightarrow Country2.prov.login \rightarrow Req1).\\
Req2 = (Country1.req.getWeather \rightarrow P1),\\
P1 \quad = (Country2.prov.getTemperature \rightarrow Country2.prov.getHumidity \rightarrow Req2\\
\qquad \mid\ Country2.prov.getHumidity \rightarrow Country2.prov.getTemperature \rightarrow Req2).\\
Req3 = Country1.req.logout \rightarrow Country2.prov.logout \rightarrow Req3).\\
\textsf{property}\ \|Goal = (Req1 \parallel Req2 \parallel Req3).
\end{array}
$$

Note that the actions are prefixed with either *Country1* or *Country2* so as to prevent synchronisations outside the mapping processes. The goal ensures that each time *C2* performs a *login* or a *logout*, then *Weather Station* eventually performs it as well. When *C2* issues a request for $getWeather$, then *Weather Station* eventually provides $getTemperature$ and $getHumidity$ in any order. Calculating the mediator amounts to computing the process M, which refines the composition $(C2 \parallel WeatherStation)$ so as to satisfy the *Goal* property.

Calvert and Lam [18] propose to calculate the composition first, then to eliminate the traces that violate the goal. Applied to our running example, first the composition $(C2 \parallel WeatherStation \parallel Goal)$ is calculated, then all the traces where the goal cannot be satisfied are removed, which results in the most general mediator called *quotient*. However, this calculation is computationally expensive as it requires covering all the trace set. In order to eliminate execution errors (e.g., deadlocks) efficiently, model checking can be used in the generation of mediators [8,19,37].

To avoid the reliance on model checking techniques, Yellin and Strom [53] propose an algorithm for the automated synthesis of mediators based on a declarative interface mapping. The authors assume a non-ambiguous one-to-one interface mapping, i.e., an operation corresponds to one operation only. They construct the mediator by exploring the protocols of the mediator and performing the necessary translations so as to guarantee that no deadlock can happen.

All the aforementioned approaches expect the transformations to be partially specified. In other words, the mediation problem has been shifted to the goal or interface mapping definition. Most of the difficulty remains on the definition of the partial specification, which require developers to know the protocols of both components and to have an intuitive understanding of the translations that need to be performed to enable them to interoperate. Given the size and the number of parameters of the interface of each component, this task may be error-prone and perhaps as difficult as providing the mediator itself. For example, the Amazon Web Service[10] includes 23 operations and no less than 72 data type definitions and eBay[11] contains more than 156 operations. Given all possible

[10] `http://soap.amazon.com/schemas2/AmazonWebServices.wsdl`

[11] `http://developer.ebay.com/webservices/latest/ebaysvc.wsdl`

combinations, methods that automatically compute this partial specification are necessary, which we survey next.

4.3 The Semantic Perspective: Emergent Protocol Mediators

Ontologies provide experts with a means to formalise the knowledge about domains as a set of axioms that make explicit the intended meaning of a vocabulary [30]. Hence, besides general purpose ontologies, such as dictionaries (e.g., WordNet[12]) and translators (e.g., BOW[13]), there is an increasing number of ontologies available for various domains such as biology [3], geoscience [44], and social networks [29], which in turn foster the development of a multitude of search engines for finding ontologies on the Web [25].

Ontologies are supported by a logic theory to reason about the properties and relations holding between the various domain entities. In particular, OWL[14] (Web Ontology Language), which is the W3C standard language to model ontologies, is based on Description Logics (DL). While traditional formal specification techniques (e.g., first-order logic) might be more powerful, DL offers crucial advantages: it excels at modelling domain-specific knowledge while providing decidable and efficient reasoning algorithms. DL is used to formally specify the vocabulary of a domain in terms of concepts, features of each concept, and relationships between these concepts [26]. DL also allows the definition of complex types out of primitive ones, is able to detect specialisation relations between complex types, and to test the consistency of types. Traditionally, the basic reasoning mechanism in DL is *subsumption*, which can be used to implement other inferences (e.g., satisfiability and equivalence) using pre-defined reductions [4]. In this sense, DL in many ways resembles type systems with some inference mechanisms such as subsumption between concepts and classification of instances within the appropriate concept, corresponding to type subsumption and type inference respectively. Nevertheless, DL is by design and tradition well-suited for application- and domain-specific services [11].

Besides defining the semantics of data, OWL-S [36] adds the definition of the *capability* of a service, which defines the service's functionality, and the service's *process model*, which defines how this functionality is performed. Services can then be matched based on their capabilities [43] or based on their process model. In this latter case, Vaculín *et al.* [51] devise a mediation approach for OWL-S processes. They first generate all requesters' paths, then find the appropriate mapping for each path by simulating the provider process. This approach deals only with client/server interactions and is not able to generate a mediator if many mappings exist for the same operation. However, OWL-S only has a qualified consent because it specifies yet another model to define services. In addition, solutions based on process algebra and automata have proven to be more suitable for reasoning about protocol interoperability.

[12] `http://www.w3.org/TR/wordnet-rdf/`
[13] `http://BOW.sinica.edu.tw/`
[14] `http://www.w3.org/TR/owl2-overview/`

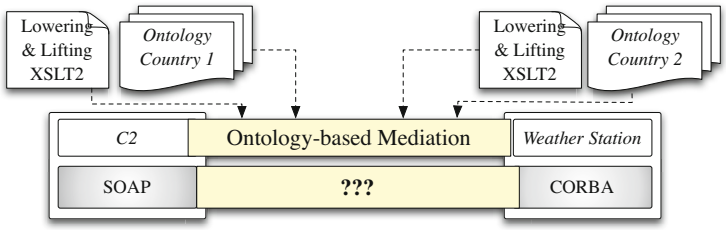


Fig. 9. Ontology-based mediation

In this direction, WSMO [20] defines a description language that integrates ontologies with state machines for representing Semantic Web Services. However, these states machines are not used to synthesise mediators. Instead, a runtime mediation framework, the Web Service Execution Environment (WSMX) mediates interaction between heterogeneous services by inspecting their individual protocols and perform the necessary translation on the basis of pre-defined mediation patterns while the composition of these patterns is not considered, and there is no guarantee that it will not lead to a deadlock.

Considering again our GMES example, with the knowledge of the weather domain encoded within a weather ontology, it can be inferred that the $getWeather$ operation required by *C2* corresponds to the $getTemperature$ and $getHumidity$ operations provided by *Weather Station*. As a result, the following mapping process can be generated:

$$Map = (req.getWeather \rightarrow prov.getTemperature \rightarrow prov.getHumidity).$$

We obtain the following connector:

$$\begin{aligned} \|\, Weather\_Connector12\_Semantic = (\ & C2\_role \\ \|\ & WeatherStation\_role \\ \|\ & (\mathsf{forall}[op : weather\_actions1]SOAPClient(\mathrm{op})) \\ \|\ & (\mathsf{forall}[op : weather\_actions1]SOAPGlue(\mathrm{op})) \\ \|\ & (\mathsf{forall}[op : weather\_actions1]SOAPServer(\mathrm{op}))) \\ \|\ & (\mathsf{forall}[op : weather\_actions2]CORBAClient(\mathrm{op})) \\ \|\ & (\mathsf{forall}[op : weather\_actions2]CORBAGlue(\mathrm{op})) \\ \|\ & (\mathsf{forall}[op : weather\_actions2]CORBAServer(\mathrm{op}))). \\ \|\ & Map \end{aligned}$$

However, *C2* and *Weather Station* cannot interact successfully through *Weather_Connector12_Semantic* since the coordination at the middleware layer is not performed. For example, it is not specified how to coordinate the $getWeather$ SOAP request with the $getTemperature$ and $getHumidity$ CORBA requests (see Figure 9).

The need for ontologies to achieve interoperability is not specific to the Web Service domain but should be considered for highly heterogeneous environments where components may be built using diverse middleware technologies. It is in particular worth highlighting the consensus that ontologies are key to the IoT vision [50]. As a result, it is indispensable to combine appropriate techniques to handle the multifaceted nature of interoperability. These techniques include formal approaches for the synthesis of mediators with support of ontology-based

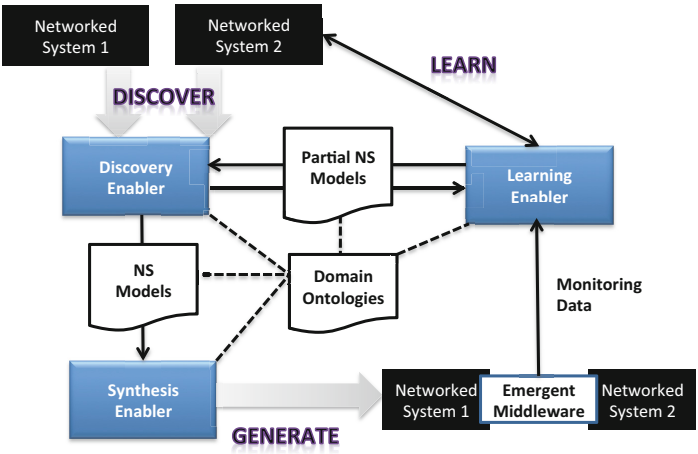


Fig. 10. The CONNECT architecture for the realisation of emergent middleware [10]

reasoning so as to automate the synthesis, together with middleware solutions to realise and execute these mediators and enable components to interoperate effectively. We have investigated such a multifaceted solution to interoperability within the CONNECT project [10].

5 Emergent Middleware: A Multifaceted Approach to Interoperability

In this section, we present the solution elaborated in the context of the European CONNECT project that revolves around the notion of emergent middleware and related enablers so as to sustain interoperability in the increasingly connected digital world. An emergent middleware is a dynamically generated distributed system infrastructure for the current operating environment and context, which allows functionally-compatible systems to interoperate seamlessly.

5.1 Emergent Middleware Enablers

In order to produce an emergent middleware, an architecture of *Enablers* is required that support the realisation of mediators into emergent middleware. An Enabler is a software component responsible for a specific step in the realisation of emergent middleware and which coordinates with other Enablers during this process.

As depicted in Figure 10, the emergent middleware Enablers are informed by *domain ontologies* that formalise the concepts associated with the application domains (i.e., the vocabulary of the application domains and their relationships) of interest as well as with middleware solutions (i.e., the vocabulary defining

middleware peculiarities, from interaction paradigms to related messages). Three *Enablers*, which are presented below, must then be comprehensively elaborated to fully realise emergent middleware.

Discovery Enabler: The *Discovery Enabler* is in charge of finding the components operating in a given environment. The Discovery Enabler receives both the advertisement messages and lookup request messages that are sent within the network environment by the components using legacy discovery protocols (e.g., SLP[15], WS-Discovery[16], UPnP-SSDP[17], Jini[18]). The Enabler obtains this input by listening on known multicast addresses (used by legacy discovery protocols), as common in interoperable service discovery [15]. These messages are then processed, using plug-ins associated with legacy discovery protocols, thereby allowing to extract basic component models from the information exposed by the components, i.e., identification of the components' interfaces together with middleware used for interactions. We build upon the ontology-based modelling as defined by Semantic Web Services (presented in Section 4.3) to model components. The model of a component includes: (i) a semantic description of the functionality it requires or provides, that is, its capability, (ii) a description of the interface of the component, which is augmented with ontology-based annotations attached to the operations required or provided by the component, (iii) a description of the interaction protocol run by the component, that is behaviour, and (iv) a specific middleware used to implement this behaviour and further refine the execution of operations. For example, as illustrated in Section 3.1, in the case of a SOAP middleware, a required operation corresponds to the sending of a SOAP request parameterised with the name of the operation, and the reception of the corresponding SOAP response. In a dual manner, a provided operation corresponds to the reception of a SOAP request parameterised with the name of the operation, and the sending of the corresponding SOAP response. However, using existing discovery protocols, components only expose their syntactic interfaces. Hence, the Discovery Enabler relies on the Learning Enabler to complete the model of a component.

Learning Enabler: The *Learning Enabler* uses advanced learning algorithms to dynamically infer the ontology-based semantics of the component's capability and operations, as well as determine the behaviour of a component, given the interface description it exposes through some legacy discovery protocol. The Learning Enabler implements both statistical and automata learning to feed component models with adequate semantic knowledge, i.e., functional and behavioural semantics, respectively [6]. The Learning Enabler must interact directly with a component in order to learn its behaviour.

[15] `http://www.openslp.org/`
[16] `http://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.pdf`
[17] `http://www.upnp.org/`
[18] `http://www.jini.org/`

Synthesis Enabler: Once component models are complete, initial semantic matching of their capabilities is performed to determine whether or not the components are functionally compatible [43]. The automated synthesis of mediators between functionally compatible components, which is the main role of the *Synthesis Enabler*, lies at the heart of the realisation of the emergent middleware. It puts together the various perspectives on interoperability presented in Section 4 so as to provide a unified solution to the automated generation of mediators and their implementation as emergent middleware.

The semantic perspective provides us with tools to compute the interface mapping automatically by using domain ontologies in order to reason about the semantics of the required and provided operations of the components and to infer the semantic correspondence between them. More specifically, we first define the conditions under which a sequence of required operations can be mapped to a sequence of provided operations. These conditions state that (i) the functionality offered by the provided operations covers that of the required ones, (ii) each provided operation has its input data available (in the right format) at the time of execution, and (iii) each required operation has its output data available (also in the appropriate format) at the time of execution. Then, we use constraint programming, which we leverage to support ontology reasoning, in order to compute the interface mapping efficiently [21].

The protocol perspective provides us with the foundations for synthesising mediators based on the generated interface mapping. More specifically, we define an approach that uses interface mapping to build the mediator incrementally by forcing the protocols at hand to progress consistently so that if one requires a sequence of operations, the interacting process is ready to engage in a sequence of provided operations to which it maps according to the interface mapping. Given that an interface mapping guarantees the semantic compatibility between the operations of the components, then the mediator synchronises with both protocols and compensates for the differences between their actions by performing the necessary transformations. The mediator further consumes the extra output actions so as to allow protocols to progress. The synthesis of mediators deals only with required and provided operations, while their actual implementation is managed by specific middleware [31].

Finally, the middleware perspective provides the background necessary to implement mediators and turn them into emergent middleware. In particular, we build upon the approach of a pairwise-mediation framework which, given a specification of the translations that need to be made, deploys the mediator and executes the necessary translations to make functionally compatible components interoperate. Hence, it suffices to provide the mediator previously synthesised as input to the mediation framework.

Adaptive Emergent Middleware: The Learning phase is a continuous process where the knowledge about components is enriched over time, thereby implying that emergent middleware possibly needs to adapt as the knowledge evolves. The synthesised emergent middleware is equipped with monitoring probes that gather information on actual interaction between connected systems. This observed *Monitoring Data*

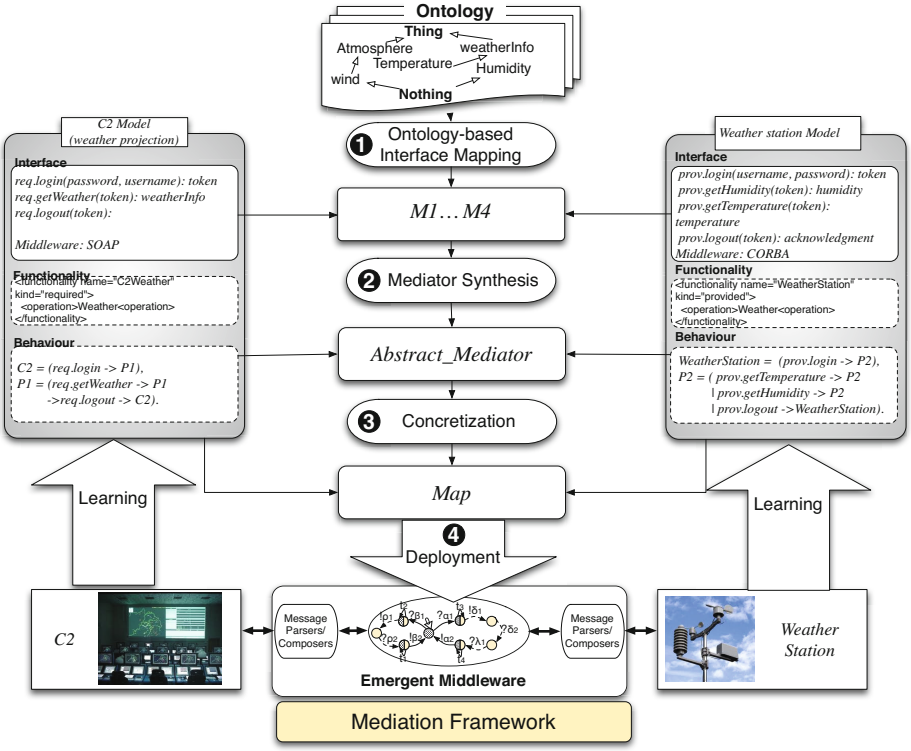


Fig. 11. Emergent middleware between *C2* and *Weather Station*

(see Figure 10) is delivered to the Learning Enabler, where the learned hypotheses about the components' behaviour are compared to the observed interactions. Whenever an observation is made by the monitoring probes that is not contained in the learned behavioural models, another iteration of learning is triggered, yielding refined behavioural models. These models are then used to synthesise and deploy an updated emergent middleware.

5.2 Emergent Middleware in GMES

Figure 11 depicts the steps to produce the emergent middleware that makes *C2* and *Weather Station* interoperate. The models of *C2* and *Weather Station* can be automatically inferred from their discovered interface as detailed in [6]. In this section, we focus on the steps for synthesising the mediator that ensures interoperability between *C2* and *Weather Station*.

Using a weather ontology, we calculate the interface mapping (see Figure 11-❶), which results in the definition of the following processes:

$$
\begin{aligned}
M1 &= (Country1.req.login \rightarrow Country2.prov.login \rightarrow \mathsf{END}).\\
M2 &= (Country1.req.getWeather \rightarrow Country2.prov.getTemperature\\
&\quad \rightarrow Country2.prov.getHumidity \rightarrow \mathsf{END}).\\
M3 &= (Country1.req.getWeather \rightarrow Country2.prov.getHumidity\\
&\quad \rightarrow Country2.prov.getTemperature \rightarrow \mathsf{END}).\\
M4 &= (Country1.req.logout \rightarrow Country2.prov.logout \rightarrow \mathsf{END}).
\end{aligned}
$$

The abstract mediator *Abstract_Map* coordinates these processes in order for the composition (*C2* ‖ *Abstract_Map* ‖ *WeatherStation*) to be free from deadlocks. When translating the *getWeather* operation required by *C2*, both $M2$ and $M3$ are applicable but we have to choose only one of them as the mediator cannot perform internal choice (see Figure 11-❷). The abstract mediator is as follows:

$$
\begin{aligned}
Abstract\_Mediator &= (Country1.req.login \rightarrow Country2.prov.login \rightarrow AMap),\\
AMap &= (Country1.req.getWeather \rightarrow Country2.prov.getTemperature\\
&\quad \rightarrow Country2.sendCORBARequest.getHumidity \rightarrow AMap\\
&\quad \mid Country1.req.logout \rightarrow Country2.prov.logout \rightarrow Abstract\_Mediator).
\end{aligned}
$$

We concretise the abstract mediator by taking into account the middleware used by each component. For example, the SOAP request for the login operation received from *C2* is translated to a CORBA request for the login operation and forwarded to *Weather Station*. Then, the associated CORBA response received from *Weather Station* is transformed to a SOAP response and sent to *C2*. A SOAP request for a *getWeather* is translated to a CORBA request for *getTemperature* together with another CORBA request for *getHumidity*. Then the CORBA responses for *getTemperature* and *getHumidity* are translated into a SOAP response for *getWeather* [5]. The resulting Map process is as follows (see Figure 11-❸):

$$
\begin{aligned}
Map &= (Country1.receiveSOAPRequest.login \rightarrow Country2.sendCORBARequest.login\\
&\quad \rightarrow Country2.receiveCORBAResponse.login \rightarrow Country1.sendSOAPResponse.login\\
&\quad \rightarrow Map1),\\
Map1 &= (Country1.receiveSOAPRequest.getWeather\\
&\quad \rightarrow Country2.sendCORBARequest.getTemperature\\
&\quad \rightarrow Country2.receiveCORBAResponse.getTemperature\\
&\quad \rightarrow Country2.sendCORBARequest.getHumidity\\
&\quad \rightarrow Country2.receiveCORBAResponse.getHumidity\\
&\quad \rightarrow Country1.sendSOAPResponse.getWeather\\
&\quad \rightarrow Map1\\
&\quad \mid Country1.receiveSOAPRequest.logout \rightarrow Country2.sendCORBARequest.logout\\
&\quad \rightarrow Country2.receiveCORBAResponse.logout \rightarrow Country1.sendSOAPResponse.logout\\
&\quad \rightarrow Map).
\end{aligned}
$$

Finally, the mediator is deployed on top of a mediation framework, Starlink [13], which executes the Map process and generates parsers and composers to deal with middleware-specific messages (see Figure 11-❹). The resulting connector *Weather_Mediator* make *C2* and *Weather Station* interoperate:

$$
\begin{aligned}
\|Weather\_Mediator = (\ & Country1 : C2\_role\\
\| \ & Country2 : WeatherStation\_role\\
\| \ & (\mathsf{forall}[op : weather\_actions1]Country1 : SOAPClient(\mathrm{op}))\\
\| \ & (\mathsf{forall}[op : weather\_actions1]Country1 : SOAPGlue(\mathrm{op}))\\
\| \ & (\mathsf{forall}[op : weather\_actions1]Country1 : SOAPServer(\mathrm{op}))\\
\| \ & (\mathsf{forall}[op : weather\_actions2]Country2 : CORBAClient(\mathrm{op}))\\
\| \ & (\mathsf{forall}[op : weather\_actions2]Country2 : CORBAGlue(\mathrm{op}))\\
\| \ & (\mathsf{forall}[op : weather\_actions2]Country2 : CORBAServer(\mathrm{op}))).\\
\| \ & Map).
\end{aligned}
$$

To sum up, this simple example allows us to illustrates the CONNECT approach to the synthesis of emergent middleware in order to achieve interoperability between components that feature differences at both the application and middleware layers. Ontologies play a crucial role in this process by allowing us to reason about the meaning of information exchanged between components and infer the mappings necessary to make them operate together. Likewise, behavioural analysis enables us to synthesise the mediator that coordinates the components' behaviours and guarantees their successful interaction. Finally, middleware technologies allow us to enact the mediator through the concept of emergent middleware. Nevertheless, to enable automated reasoning about interoperability and the generation of appropriate mediators, we focus on the ontological concepts, which represent the types of the input/output data exchanged between components. However, there are situations where reasoning about the value of the data is also necessary. For example, to be able to ensure interoperability between components using different streaming protocols, the mediator is required to deal with lower-level details such as the appropriate encoding and the segmentation of data. In this particular case, the mediator can be partially specified at design-time and deployed at runtime, as is the case for AmbiStream [2].

The accuracy of the components' models may also impact the emergent middleware. While machine learning significantly improves automation by inferring the model of the component from its implementation, it also induces some inaccuracy that may lead the emergent middleware to reach an erroneous state. Hence, the system needs to be continuously monitored so as to evaluate the correspondence between the actual system and its model. In the case where the model of one of the components changes, then the mediator should be updated accordingly in order to reflect this change. Another imprecision might also be due to ontology alignment. Hence, incremental re-synthesis would be very important to cope with both the dynamic aspect and partial knowledge about the environment.

6 Conclusion

In spite of the extensive research effort, interoperability remains an open and critical challenge for today's and even more tomorrow's highly heterogeneous and dynamic networking environments. This chapter has surveyed state-of-the-art approaches to interoperability, highlighting the multiple perspectives that need to be considered and which span: (i) middleware-layer implementation so as to provide abstractions hiding the heterogeneity of the environment, (ii) protocol synthesis so as to relieve as much as possible the developers in dealing with the implementation of custom mediators that must overcome heterogeneity from the application down to the middleware layers, and (iii) ontology-based specification of system models so as to allow fully automated mediator synthesis that is a key requirement of the dynamic networking environment. The chapter has then outlined the CONNECT approach to interoperability, which unifies these different perspectives so as to enable interoperability in a future-proof manner. CONNECT

specifically advocates a solution based on maintaining a sophisticated model of the system at runtime [5]. This includes capturing aspects related to the components' capabilities, interfaces, associated data and behaviour. The solution is then supported by a range of enablers that capture or act on this information to enable the runtime realisation of emergent middleware between given components, i.e., protocol mediators that reconcile the discrepancies occurring in both application- and middleware-layer protocols. The end result can be seen as a two dimensional space related to (i) the meta-information that is captured about the system, and (ii) the associated middleware functions that operate on this meta-information space. This is then supported by ontologies that provide meaning and reasoning capabilities across all enablers and aspects of meta-information known about the system.

It is important to stress that interoperability is, as with many features of distributed systems, an end-to-end problem. For example, it is not sufficient to achieve interoperability between application-level interfaces. Rather, interoperability can only be achieved through a coordinated approach involving application, middleware and underlying network levels so that components can interoperate in spite of potential heterogeneity in descriptions, in middleware deployments and network environments they operate in. And, this needs to be achieved dynamically according to the current context, assuming contexts may change, thereby requiring self-adaptive emergent middleware. More generally, future work includes examining the application of the CONNECT approach to deal with uncontrolled changes in the environment, and expanding the scope of the work to include non-functional concerns associated with communication instances (including performance, dependability and security properties). There is also considerable potential for core research on emergent middleware in areas such as the role of probabilistic reasoning in order to support uncertainties in the ontology, the possibility of learning new ontological information as it becomes available, and also dealing with heterogeneity in the ontologies.

Acknowledgments. This work is carried out as part of the European FP7 ICT FET CONNECT (`http://connect-forever.eu/`) project.

References

1. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Trans. Softw. Eng. Methodol. (1997)
2. Andriescu, E., Speicys Cardoso, R., Issarny, V.: AmbiStream: A middleware for multimedia streaming on heterogeneous mobile devices. In: Kon, F., Kermarrec, A.-M. (eds.) Middleware 2011. LNCS, vol. 7049, pp. 249–268. Springer, Heidelberg (2011)
3. Aranguren, M., Bechhofer, S., Lord, P., Sattler, U., Stevens, R.: Understanding and using the meaning of statements in a bio-ontology: recasting the gene ontology in OWL. BMC Bioinformatics 8(1), 57 (2007)
4. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.: The Description Logic Handbook. Cambridge University Press (2003)

5. Bencomo, N., Bennaceur, A., Grace, P., Blair, G., Issarny, V.: The role of models@run.time in supporting on-the-fly interoperability. Springer Journal on Computing (2012)
6. Bennaceur, A., Issarny, V., Sykes, D., Howar, F., Isberner, M., Steffen, B., Johansson, R., Moschitti, A.: Machine learning for emergent middleware. In: Proc. of the Joint Workshop on Intelligent Methods for Soft. System Eng., JIMSE (2012)
7. Bennaceur, A., Issarny, V., Spalazzese, R., Tyagi, S.: Achieving interoperability through semantics-based technologies: The instant messaging case. In: Cudré-Mauroux, P., et al. (eds.) ISWC 2012, Part II. LNCS, vol. 7650, pp. 17–33. Springer, Heidelberg (2012)
8. Bersani, M., Cavallaro, L., Frigeri, A., Pradella, M., Rossi, M.: SMT-based verification of ltl specification with integer constraints and its application to runtime checking of service substitutability. In: 2010 8th IEEE International Conference on Software Engineering and Formal Methods (SEFM), pp. 244–254. IEEE (2010)
9. Blair, G.S., Paolucci, M., Grace, P., Georgantas, N.: Interoperability in complex distributed systems. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 1–26. Springer, Heidelberg (2011)
10. Blair, G.S., Bennaceur, A., Georgantas, N., Grace, P., Issarny, V., Nundloll, V., Paolucci, M.: The role of ontologies in emergent middleware: Supporting interoperability in complex distributed systems. In: Kon, F., Kermarrec, A.-M. (eds.) Middleware 2011. LNCS, vol. 7049, pp. 410–430. Springer, Heidelberg (2011)
11. Borgida, A.: From type systems to knowledge representation: Natural semantics specifications for description logics. Int. J. Cooperative Inf. Syst. 1(1), 93–126 (1992)
12. Bromberg, Y.-D.: Solutions to middleware heterogeneity in open networked environment. Ph.D. thesis, Université de Versailles Saint-Quentin-en-Yvelynes (2006)
13. Bromberg, Y.-D., Grace, P., Réveillère, L.: Starlink: Runtime interoperability between heterogeneous middleware protocols. In: International Conference on Distributed Computing Systems, ICDCS (2011)
14. Bromberg, Y.-D., Grace, P., Réveillère, L., Blair, G.S.: Bridging the interoperability gap: Overcoming combined application and middleware heterogeneity. In: Kon, F., Kermarrec, A.-M. (eds.) Middleware 2011. LNCS, vol. 7049, pp. 390–409. Springer, Heidelberg (2011)
15. Bromberg, Y.-D., Issarny, V.: INDISS: Interoperable discovery system for networked services. In: Alonso, G. (ed.) Middleware 2005. LNCS, vol. 3790, pp. 164–183. Springer, Heidelberg (2005)
16. Bromberg, Y.-D., Réveillère, L., Lawall, J.L., Muller, G.: Automatic generation of network protocol gateways. In: Bacon, J.M., Cooper, B.F. (eds.) Middleware 2009. LNCS, vol. 5896, pp. 21–41. Springer, Heidelberg (2009)
17. Calero, C., Ruiz, F., Piattini, M.: Ontologies for Software Engineering and Software Technology. Springer (2006)
18. Calvert, K.L., Lam, S.S.: Deriving a protocol converter: A top-down method. In: Proc. of the Symposium on Communications Architectures & Protocols, SIGCOMM, pp. 247–258 (1989)
19. Cavallaro, L., Di Nitto, E., Pradella, M.: An automatic approach to enable replacement of conversational services. In: Baresi, L., Chi, C.-H., Suzuki, J. (eds.) ICSOC-ServiceWave 2009. LNCS, vol. 5900, pp. 159–174. Springer, Heidelberg (2009)
20. Cimpian, E., Mocan, A.: WSMX process mediation based on choreographies. In: Bussler, C.J., Haller, A. (eds.) BPM 2005. LNCS, vol. 3812, pp. 130–143. Springer, Heidelberg (2006)

21. Connect Consortium: Connect Deliverable D3.3: Dynamic connector synthesis: Revised prototype implementation. FET IP Connect EU project, http://hal.inria.fr/hal-00695592/
22. Connect Consortium: Connect Deliverable D6.3: Experiment scenarios, prototypes and report - Iteration 2. FET IP Connect EU project, http://hal.inria.fr/hal-00695639
23. Connect Consortium: Connect Deliverable D6.4: Assessment report: Experimenting with CONNECT in Systems of Systems, and Mobile Environments. FET IP Connect EU project, http://hal.inria.fr/hal-00793920
24. Coulouris, G.F., Dollimore, J., Kindberg, T., Blair, G.: Distributed systems: concepts and design, 5th edn. Addison-Wesley, Longman (2012)
25. d'Aquin, M., Noy, N.F.: Where to publish and find ontologies? a survey of ontology libraries. J. Web Sem. 11, 96–111 (2012)
26. Dong, J.S.: From semantic web to expressive software specifications: a modeling languages spectrum. In: Proc. of the International Conference on Software Engineering, ICSE (2006)
27. Garlan, D., Allen, R., Ockerbloom, J.: Architectural mismatch or why it's hard to build systems out of existing parts. In: International Conference on Software Engineering, ICSE (1995)
28. Georgantas, N., Rahaman, M.A., Ameziani, H., Pathak, A., Issarny, V.: A coordination middleware for orchestrating heterogeneous distributed systems. In: Riekki, J., Ylianttila, M., Guo, M. (eds.) GPC 2011. LNCS, vol. 6646, pp. 221–232. Springer, Heidelberg (2011)
29. Golbeck, J., Rothstein, M.: Linking social networks on the web with foaf: A semantic web case study. In: AAAI, pp. 1138–1143 (2008)
30. Guarino, N.: Helping people (and machines) understanding each other: The role of formal ontology. In: Meersman, R., Tari, Z. (eds.) CoopIS/DOA/ODBASE 2004, Part 1. LNCS, vol. 3290, p. 599. Springer, Heidelberg (2004)
31. Issarny, V., Bennaceur, A., Bromberg, Y.-D.: Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 217–255. Springer, Heidelberg (2011)
32. Jamshidi, M.: Systems of systems engineering: principles and applications. CRC Press (2008)
33. Keller, R.M.: Formal verification of parallel programs. Communications of the ACM 19(7), 371–384 (1976)
34. Lam, S.S.: Protocol conversion. IEEE Transaction Software Engineering (1988)
35. Magee, J., Kramer, J.: Concurrency: State models and Java programs. Wiley, Hoboken (2006)
36. Martin, D.L., Burstein, M.H., McDermott, D.V., McIlraith, S.A., Paolucci, M., Sycara, K.P., McGuinness, D.L., Sirin, E., Srinivasan, N.: Bringing semantics to web services with owl-s. In: Proc. of the World Wide Web Conference, WWW 2007, pp. 243–277 (2007)
37. Mateescu, R., Poizat, P., Salaün, G.: Adaptation of service protocols using process algebra and on-the-fly reduction techniques. IEEE Trans. Software Eng. 38(4), 755–777 (2012)
38. McIlraith, S.A., Son, T.C., Zeng, H.: Semantic web services. IEEE Intelligent Systems 16(2), 46–53 (2001)
39. Menge, F.: Enterprise Service Bus. In: Proc. of the Free and Open Source Soft. Conf. (2007)

40. Mokhtar, S.B., Georgantas, N., Issarny, V.: Cocoa: Conversation-based service composition in pervasive computing environments with qos support. Journal of Systems and Software 80(12), 1941–1955 (2007)
41. Ben Mokhtar, S., Kaul, A., Georgantas, N., Issarny, V.: Efficient semantic service discovery in pervasive computing environments. In: van Steen, M., Henning, M. (eds.) Middleware 2006. LNCS, vol. 4290, pp. 240–259. Springer, Heidelberg (2006)
42. Nitto, E.D., Rosenblum, D.S.: Exploiting adls to specify architectural styles induced by middleware infrastructures. In: Proc. of International Conference on Software Engineering, ICSE (1999)
43. Paolucci, M., Kawamura, T., Payne, T.R., Sycara, K.: Semantic matching of web services capabilities. In: Horrocks, I., Hendler, J. (eds.) ISWC 2002. LNCS, vol. 2342, pp. 333–347. Springer, Heidelberg (2002)
44. Raskin, R.G., Pan, M.J.: Knowledge representation in the semantic web for earth and environmental terminology (SWEET). Computers & Geosciences 31(9), 1119–1125 (2005)
45. Shadbolt, N., Berners-Lee, T., Hall, W.: The semantic web revisited. IEEE Intelligent Systems 21(3), 96–101 (2006)
46. Shaw, M.: Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. In: Lamb, D.A. (ed.) ICSE-WS 1993. LNCS, vol. 1078, pp. 17–32. Springer, Heidelberg (1996)
47. Spitznagel, B., Garlan, D.: A compositional formalization of connector wrappers. In: International Conference on Software Engineering, ICSE (2003)
48. Tanenbaum, A., Van Steen, M.: Distributed systems: principles and paradigms, 2nd edn. Prentice Hall (2006)
49. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software architecture: foundations, theory, and practice. Wiley, Hoboken (2009)
50. Uckelmann, D., Harrison, M., Michahelles, F.: Architecting the internet of things. Springer (2011)
51. Vaculín, R., Neruda, R., Sycara, K.P.: The process mediation framework for semantic web services. International Journal of Agent-Oriented Software Engineering, IJAOSE 3(1), 27–58 (2009)
52. Wiederhold, G.: Interoperation, mediation, and ontologies. In: Proc. of the Fifth International Symposium on Generation Computer Systems Workshop on Heterogeneous Cooperative Knowledge-Bases, pp. 33–48. Citeseer (1994)
53. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM Trans. Program. Lang. Syst. (1997)

Controlling Application Interactions on the Novel Smart Cards with Security-by-Contract

Olga Gadyatskaya and Fabio Massacci

DISI, University of Trento,
via Sommarive, 14, Povo 0, Trento, Italy, 38123
name.surname@unitn.it

Abstract. In this paper we investigate novel use cases for open multi-application smart card platforms. These use cases require a fine-grained access control mechanism to protect the sensitive functionality of on-card applications. We overview the Security-by-Contract approach that validates at load time that the application code respects the interaction policies of other applications already on the card, and discuss how this approach can be used to address the challenging change scenarios in the target use cases.

1 Introduction

The smart card technology supports asynchronous coexistence of multiple applications from different providers on the same chip since a long time ago. However, the actual use cases for such cards have appeared only recently. In this paper we briefly overview two novel use cases for multi-application smart cards, which we explore as illustrative examples. The first use case is the Near Field Communication (NFC)-enabled smartphone, where the (U)SIM card hosts payment, transport and other types of sensitive applications coming from different vendors. The second use case is a smart card-based enhancement of a smart meter system. A telecommunications hub, implemented as a smart card and installed at a house, hosts and manages traditional utility consumption applications, such as gas or electricity consumption applications, and also a set of telecare applications. The telecare applications enable remote monitoring of health status of the inhabitants; they are connected to devices such as weights or a heart rate monitor. This architectural solution was developed within the Hydra Project [12].

In both these scenarios the applications deployed on the multi-application smart cards are quite sensitive, as they may have access to the private data of the device owner or the application provider. However, these applications are not necessarily fully sandboxed. On the contrary, the applications might need to interact with each other on the card in order to provide an enhanced functionality to the device owner. Thus the key challenge for such cards it ensuring that only trusted partners will have access to the shared functionality (called *service* in the smart card jargon).

E. Giachino et al. (Eds.): FMCO 2012, LNCS 7866, pp. 197–215, 2013.

The existing solutions for control of application interactions on multi-tenant platforms mostly propose to verify of a pre-defined set of applications off-card [2] or enforce the desired policies at run-time [1]. The first approach is not appealing from the business perspective: as the platform is open, each time a new application will be loaded a full offline re-verification will be required. The second approach is simply not suitable for smart cards due to the resource constraints. The approach that is currently adopted by the smart card community is embedding the access control checks into the functional code. Each time the sensitive service is invoked it checks that the caller is authorized to use it. However, this approach suffers from the fact that partial code updates are not available on smart cards; only the full reinstallation is supported by the runtime environment. Therefore each time a new trusted caller needs to be added a full reinstallation of the application will be required.

Recently load time verification was adopted for multi-application smart cards [4,5,8,6,7]. With this approach the platform is always in a secure state across all possible changes, such as loading of a new application or removal of an old one. In the current paper we overview the Security-by-Contract (S×C) approach for load time verification on multi-application Java Cards and identify how the novel multi-application smart card use cases can be handled with this approach.

The paper is structured is follows. We overview the target use cases in Sec. 2 and present the workflows of the S×C approach for each change scenario in Sec. 3. The necessary background on Java Card is presented in Sec. 4, the design details of the framework are summarized in Sec. 5, and the concrete contracts for the identified motivational scenarios are listed in Sec. 6. We overview the related work in Sec. 7 and conclude in Sec. 8.

2 Multi-application Java Card Use Cases

In this section we present the target use cases recently introduced in the multi-application smart cards domain.

2.1 NFC-Enabled Phones

Currently NFC offerings from various vendors include *payment* applications (the Google wallet[1], PayPass from MasterCard[2], payWave from VISA[3]), *ticketing* applications (Calypso is a set of technical specifications for NFC ticketing; its handbook contains an overview of the NFC ticketing status in various countries[4]) and *entertainment* applications (including NFC tap-triggered messages from Santa Claus[5]).

[1] http://www.google.com/wallet/
[2] http://www.paypass.com/
[3] http://www.paypass.com/
[4] http://www.calypsonet-asso.org/downloads/100324-CalypsoHandbook-11.pdf
[5] http://www.nfcworld.com/2012/12/07/321480/christmas-app-conjures-up-santa-with-an-nfc-tap/

The NFC functionality requires a secure element to store the sensitive NFC credentials, and one of the existing solutions is usage of the (U)SIM card within the phone to store this data. For instance, an NFC-enabled multi-application (U)SIM card, called UpTeq[6], is currently offered by Gemalto. This card is certified by the VISA, MasterCard and Amex payment systems.

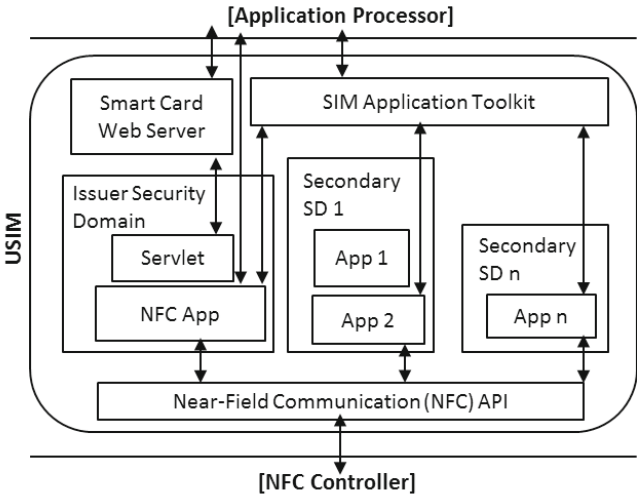


Fig. 1. (U)SIM as a secure element [14]

Figure 1 presents an architecture of a (U)SIM card used as a secure element within an NFC-enabled phone. The NFC controller enables a communication link between the phone and various NFC tags and devices.

Scenario 1: New Application Is Loaded. We consider the following scenario of an NFC-enabled smartphone with a (U)SIM-based secure element. The scenario is purely fictional scenario and we use real commercial product names only for the sake of clarity. Two applications (*applets* for short) are already hosted by the (U)SIM card: the payment application payWave from VISA and the ticketing application Touch&Travel from Deutsche Bahn[7]. The phone holder can use the payWave application for executing payment operations in shops, and the Touch&Travel application to pay for train tickets and display ticket barcodes to the phone holder and the train authorities. The VISA consortium and Deutsche Bahn are business partners, and therefore the applications can interact on card: Touch&Travel relies on payWave for the ticket payments. The (U)SIM card is managed by a telecom operator, which has agreements with both VISA and

[6] http://www.gemalto.com/telecom/upteq/index.html
[7] https://www.touchandtravel.de/

Deutsche Bahn; these agreements do not limit the telecom operator in which other applications can be loaded on the card, provided the operator guarantees that only authorized applications will interact with payWave and Touch&Travel.

The phone holder travels from Germany to France, where she installs the public ticketing application Navigo[8] produced by the French public transportation organization STIF. STIF and VISA do not have an agreement, therefore the Navigo application can be recharged only at the metro stations, and not through payWave. However, the telecom operator still has to ensure that Navigo will not try to access payWave directly on the card. In the next sections we will discuss how this can be implemented.

2.2 Communication Hubs within the Smart Grids

Our second motivating use case for multi-application smart cards is the telecommunications hub within the smart metering system proposed by the Hydra project [12]. The project aims to introduce remote care services as an extension to the smart metering system. The architecture of this extension is presented in Fig. 2. Existing utility meters and telecare devices, such as blood pressure monitor or heart-rate monitor, are connected to the smart card, which acts as a proxy between the metering devices and the corresponding utility/telecare providers.

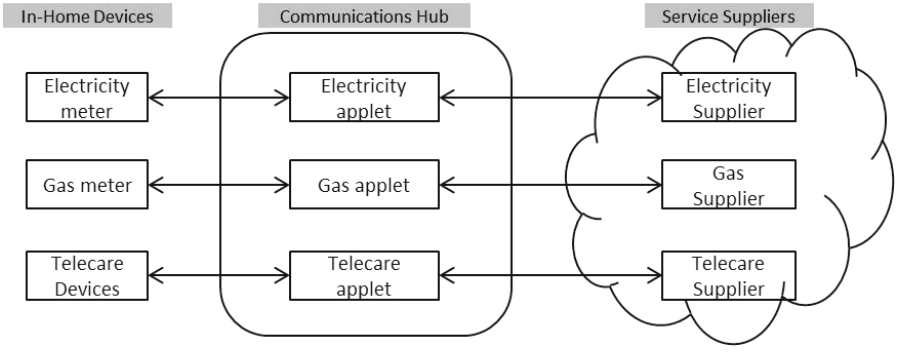


Fig. 2. Extension of a smart metering system with telecare services [12]

The main idea behind the smart card utilization is privacy of the utility consumption data. The smart metering systems measure the utility consumption at high granularity. By gathering a lot of data points throughout the day the utility companies can learn a lot about the private life of their customers: when they leave to work and come back, when they wake up and go to sleep. In an industrial setting the utility consumption can reveal details of the production process: what machinery is used, or when a new process is adopted [13]. The fact

[8] `http://test.navigo.fr/`

that the utility companies can leak this private data to third parties is even more disturbing [17]. Solutions to this privacy problem in the smart grid have started to emerge, focusing on introducing a mediator for the utility consumption or devising privacy-preserving protocols among the utility provider, the user and the meter. The mediator (for instance, a battery in case of the energy consumption) can obfuscate the actual consumption of the house owner by withdrawing the energy from the grid in a manner that would be probabilistically-independent of the actual consumption [13]. In contrast, with the privacy-preserving protocols the user will compute the utility bill from the actual utility usage and transmit this bill to the provider alongside a zero-knowledge proof that ensures the calculation to be correct and leaks no utility consumption calculation [15].

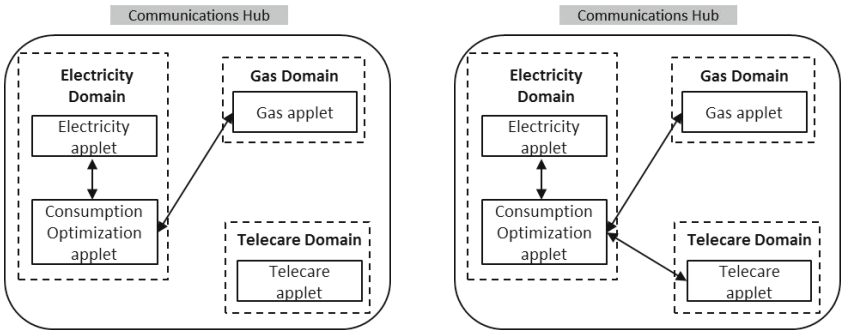
Similarly to the privacy-preserving protocols idea, in the smart meter system architecture proposed in the Hydra project the utility consumption is computed and billed directly on the smart card; afterwards the billing is displayed to the customer via the standard web interface. Therefore the private consumption data does not leave the metering system, and the utility provider can only see the total amount of the consumed utility. The Hydra architecture invites multiple utility providers to share the platform; this is possible with the multi-application smart card solution. The secure communication channels are established between a utility meter, the corresponding application on the card and the provider. These channels are available due the GlobalPlatform middleware present on the card.

GlobalPlatform is a set of card and device specifications produced and maintained by the GlobalPlatform consortium[9]. The specifications identify interoperability and security requirements for development, deployment and management of smart cards. However, the application interactions are not controlled by GlobalPlatform; they are managed by the Java Card run-time environment (JCRE), which we will further overview.

Scenario 2: Existing Application Updates Its Policy. We refine the scenario in Fig. 2 and introduce an additional Consumption Optimization application from the electricity provider. This application is connected to the household appliances in order to manage the electricity consumption in a cost-efficient manner. For instance, this application will turn on the washing machine only in the evenings when the electricity cost is the lowest.

The Consumption Optimization applet receives data on the current electricity consumption from the Electricity applet; therefore these applets interact directly on the hub. We consider that the gas provider would also like to provide the optimization services for the customer: he would like to manage the gas consumption (for instance, for heating) in a cost-efficient manner. For this purpose the Consumption Optimization applet can be used, because it is already connected to all the appliances. The electricity and the gas providers sign an agreement, and the interaction between the Gas applet and the Consumption Optimization applet is established on the hub. We emphasize that our focus in this illustrative scenario is authorization of applications for interactions; we do not consider the privacy

[9] `www.globalplatform.org`



(a) Allowed Applet Interactions Before Policy Update

(b) Allowed Applet Interactions After Policy Update

Fig. 3. Application Interactions on the Smart Meter System Communications Hub

concerns that potentially arise from the interaction of the Gas and Consumption Optimization applets. The privacy problem must be handled by the providers separately.

Initially the telecare provider and the electricity provider do not have an agreement; therefore, their applets cannot interact. However, later the electricity provider might be interested in lending the Consumption Optimization applet services also to the Telecare provider, and he will allow the interaction. Fig. 3 summarizes the authorized interactions before and after this policy update of the Consumption Optimization applet.

Notice, that the standard smart card application update procedure is full deletion of the old version and loading of the new one; partial code updates are not supported. In this setting the cost of adding a single authorization is quite significant: for some cards each new code version needs to be agreed with the platform controlling authority or the application provider himself does not have a code loading privilege and has to request the entity with this privilege to perform the code loading. Therefore the provider would like to execute the policy update independently, using the standard protocols for communication with the platform. In the S×C approach we enable the providers with this option.

3 The Security-by-Contract Components and Workflows

The illustrative use cases and scenarios presented in §§2.1-2.2 identify the need of the smart card system to provide access control facilities for applet interaction. We propose the S×C approach for multi-application smart cards to enable the applet authorization and validate the applet code with respect to the interaction policies at load time. This approach ensures that the platform is secure with respect to the applet interactions across the platform changes: installation of a new applet, removal or update of an old one. The main components of the S×C framework are the ClaimChecker, the PolicyChecker and the PolicyStore, their responsibilities are specified for each type of the platform change.

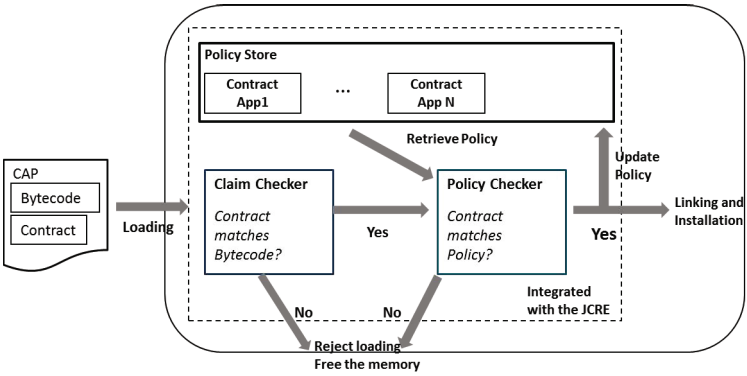


Fig. 4. The S×C workflow for load time validation

The schema for load time validation is presented in Fig. 4. The application provider delivers on the platform the applet code together with the *contract*. The contract specifies the *claimed interactions* (the services that are provided in this applet and the services of other applets that the applet may invoke at run-time) and the *policy of this application* (which applets are authorized to interact with its services and the necessary services from other applets). Notice that the service deemed necessary for some application have to be a subset of the services called by this application. The first step of load time validation on the card is the check that the contract is compliant with the code; this check is executed by the ClaimChecker component. The second step is matching the contract and the *platform policy*, which is composed by the contracts of all currently installed applications; this check is performed by the PolicyChecker component that retrieves the platform policy from the PolicyStore. If both steps are successful, the S×C admits this applet on the card, and the contract of this applet is added to the platform policy. If any of the checks failed, the loading process will be aborted and this applet will not be admitted to the platform.

For the case of the applet deletion from the platform, the S×C framework has to check that the platform will be secure with respect to application interactions once the requested deletion is performed. We present the workflow for removal in Fig. 5. In the removal workflow only the PolicyChecker and the PolicyStore components are invoked; the ClaimChecker is not required. For the application policy update scenario (presented in Fig. 6) the PolicyChecker has to ensure that after the update the platform will be in the secure state. In this case the ClaimChecker is not invoked, because the code-contract compliance was already validated at the installation step. The S×C workflow for update is designed only for the policy updates; it cannot handle the code updates, which have to be executed throughout the standard smart card code loading process.

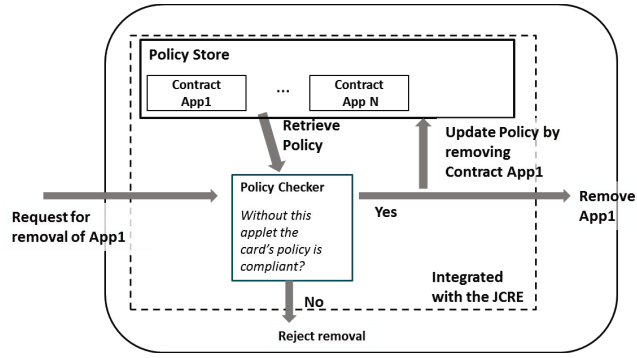


Fig. 5. The S×C workflow for load time validation

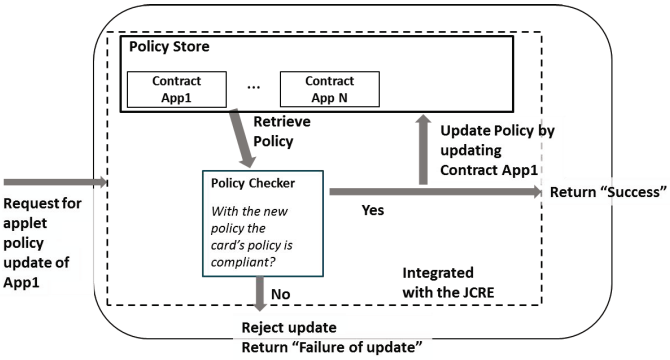


Fig. 6. The S×C workflow for applet policy update validation

4 A Primer on the Java Card Technology

Before describing the concrete contract and policy implementation and the S×C framework design details, we provide a necessary background on Java Card.

The Java Card platform architecture consists of several layers that include device hardware, a proprietary embedded operating system (Native OS), the JCRE and the installed applications [18]. The JCRE comprises multiple components, some of them belong to the Java Card interface and other belong to the Native interface. The Java Card interface components are the Installer (the entity responsible for the loading and installation processes, it is exposed for communications from the terminal, which is an external device that powers the card up and communicates with it) and the Java Card API; these parts are written in Java Card (subset of Java) and can allocate the EEPROM memory (the persistent modifiable memory).

The Native interface components are the Java Card Virtual Machine (JCVM), the Loader (entity responsible for processing the delivered CAP files) and the

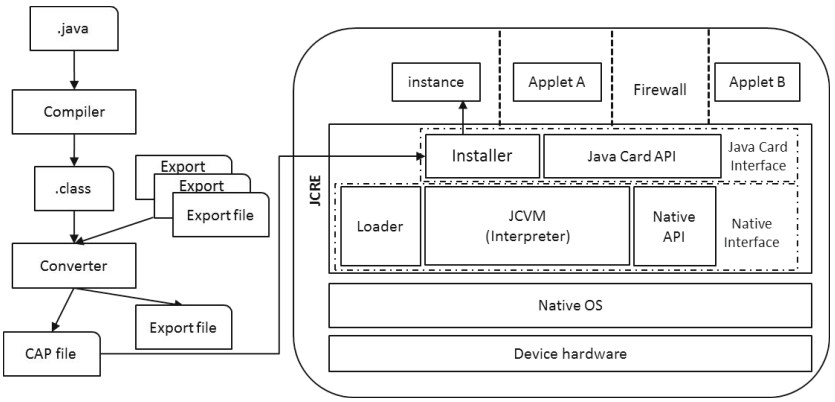


Fig. 7. The Java Card architecture and the applet deployment process

Native API. These components are typically written in the native code (C); they are printed in ROM (non-modifiable persistent memory) and are not allowed to allocate the EEPROM memory.

An application package is written in Java, compiled into a set of class files and converted into a CAP (Converted APplet) file, which is delivered on the card. CAP files are optimized by the Converter in order to occupy less space. For instance, a CAP file contains a single Constant Pool component, a single Method component with the full bytecode set of all methods defined in this package, etc. A quite similar approach for optimization of packages loaded on the device is adopted on the Android platform[10]. Each package can contain multiple applications, but interactions inside a package cannot be regulated. Also, each package is loaded in a single pass, and it is not possible to add a malicious applet to the package of an honest applet. Therefore in the sequel we consider that each package contains exactly one application and use words package and applet interchangeably.

The CAP file is transmitted onto a smart card, where it is processed, linked and then an application instance can be created. One of the main technical obstacles for the verifier running on Java Card is unavailability of the application code for reverification purposes after linking. Thus the application policy cannot be stored within the application code itself, as the verifier will not have access to it later.

The Java Card platform architecture and the applet deployment process are summarized in Fig. 7.

4.1 Application Interactions

Applications on Java Card are separated by a firewall, and the interactions between applets from different packages are always mediated by the JCRE.

[10] `www.android.com`

If two applets belong to different packages, they belong to different *contexts*. The Java Card firewall confines applet's actions to its designated context. Thus an applet can freely reach only objects belonging to its own context. The only objects accessible through the firewall are methods of *shareable interfaces*, also called *services*. A shareable interface is an interface that extends `javacard.framework.Shareable`.

An applet A implementing some services is called a *server*. An applet B that tries to call any of these services is called a *client*. A typical scenario of service usage starts with a client's request to the JCRE for a reference to A's object (that is implementing the necessary shareable interface). The firewall passes this request to application A, which decides if the reference can be granted or not based on its access control rules. The current method for services access control implementation on Java Card is the list of trusted clients embedded into the applet code. The caller can be identified through the Java Card `getPreviousContext` API. If the decision is positive, the reference is passed through the firewall and the client can now invoke any method declared in the shareable interface which is implemented by the referenced object. During invocation of a service a *context switch* will occur, thus allowing invocation of a method of the application A from a method of the application B. A call to any other method, not belonging to the shareable interface, will be stopped by the Java Card firewall; while the calls from A to its own methods or the JCRE entry points are allowed by the firewall rules [18]. Notice that with the S×C framework on board the access control policies embedded into the applet code will become obsolete.

In order to realize the interaction scenario the client has necessarily to import the shareable interface of the server and to obtain the *Export file* of the server, that lists shared interfaces and services and contains their *tokens*. Tokens are used by the JCRE for linking on the card similarly as Unicode strings are used for linking in standard Java class files. A service s can be uniquely identified as a tuple $\langle A, I, t\rangle$, where A is a unique application identifier (AID) of the package that provides the service s, that is assigned according to the standard ISO/IEC 7816-5, I is a token for a shareable interface where the service is defined and t is a token for the method in the interface I. Further, in case the origin of the service is clear, we will omit the AID and will refer to a service as a tuple $\langle I, t\rangle$.

The server's Export file is necessary for conversion of the client's package into a CAP file. In a CAP file all methods are referred to by their tokens, and during conversion from class files into a CAP file the client needs to know correct tokens for services it invokes from other applications. Shareable interfaces and Export files do not contain any implementation, therefore it is safe to distribute them.

5 Design and Implementation Details

Let us present the details of the S×C framework implementation.

5.1 Contracts

The contract of an application is defined as follows. AppClaim of an application specifies provided (the Provides set) and invoked (the Calls set) services. Let A be an application and $A.s$ be its service. We say that the service $A.s$ is *provided* if applet A is loaded and it has service s. Service $B.m$ is *invoked* by applet A if A may try to call $B.m$ during its execution. The AppClaim will be verified for compliance with the bytecode (the CAP file) by the ClaimChecker.

The application policy AppPolicy contains *authorizations for services access* (the sec.rules set) and *functionally necessary services* (the func.rules set). We say a service is necessary if a client will not be functional without this service on board. The AppPolicy lists applet's requirements for the smart card platform and other applications loaded on it.

Thus the application contract is: $\mathsf{Contract} = \langle \mathsf{AppClaim}, \mathsf{AppPolicy} \rangle$, where $\mathsf{AppClaim} = \langle \mathsf{Provides}, \mathsf{Calls} \rangle$ and $\mathsf{AppPolicy} = \langle \mathsf{sec.rules}, \mathsf{func.rules} \rangle$.

Writing and Delivering Contracts. A provided service is identified as a tuple $\langle A, I, s \rangle$, where A is the AID of package A (as the AID of A is explicit in the package itself, it can be omitted from the provided service identification tuple), and I and s are the tokens of the shareable interface and the method that define the service. Correspondingly, a called service can be identified as a tuple $\langle B, I, s \rangle$, where B is the AID of the package containing the called service.

An authorization rule is a tuple $\langle B, I, s \rangle$, where B is the AID of package B that is authorized to access the provided service with interface token I and method token s. Notice that A is not specified in its authorization rules because the rules are delivered within the A's package and the service provider is implicitly identified. Later, when the authorization rules will be added to the platform security policy, the service provider will be explicitly specified.

$\mathsf{func.rules_A}$ is a set of functionally necessary services for A, we consider that without these services provided, an application A cannot be functional. Thus we can ensure availability of necessary services. A functionally necessary service can be identified in the same way as a called service. Moreover, we insist that $\mathsf{func.rules_A} \subseteq \mathsf{Calls_A}$, as we cannot allow to declare arbitrary services as necessary, but only the ones that are at least potentially invoked in the code.

The provided service tokens can be found in the Export file, where the fully-qualified interface names and method names are present. The called services can be determined from the Export files of the desired server applets, which are consumed for conversion. More details on the token extraction from the Export files can be found in [6,7]. However, the contract-code matching will be performed on card with the CAP file, as the Export files are not delivered on board.

An important problem is contract delivery on the platform. The Java Card specification, besides the standard CAP file components, allows Custom components to be present in CAP files [18]. We have organized the contract delivery within a specific Contract Custom component. In this way the contract can be securely attached to the bytecode and sealed by the provider's signature. The standard Java Card tools do not include means to provide Custom components,

so for the proof-of-concept implementation we have designed a simple CAP Modifier tool with a GUI that provides means to write contracts and add them as a Custom component to standard CAP files. More details are available in [7].

5.2 The S×C Components

We now specify the design of each component of the S×C framework.

The ClaimChecker. The ClaimChecker is responsible for the contract-code matching step. It has to retrieve the contract from the Custom component and verify that the AppClaim is faithful (the application policy part is not matched with the code). For every service from the Provides set the ClaimChecker will find its declaration in the Export component of the CAP file; and it will ensure no undeclared services are present in the Export component. For every called service from the Calls set the ClaimChecker will identify the point in the code when this service is invoked.

Specifically, the ClaimChecker will parse the CAP file and find all the service invocation instructions (the `invokeinterface` opcode). From the operands of this instruction we can identify the invoked method token and the pointer to the Constant Pool component, from which we can resolve the needed invoked interface token and the AID of the called applet. Concrete details of this procedure are available in [7]. To implement the CAP file processing the ClaimChecker has to be integrated with the platform Loader component, as only this component has direct access to the loaded code.

The PolicyChecker. The PolicyChecker component is responsible for ensuring compliance of the platform security policy and the contract for the loading protocol. Namely, it will check that 1) for all called services from Calls, their providers have authorized the interaction in the contracts; 2) for any provided service from Provides if there is some applet on the platform that can try to invoke this service, there is a corresponding authorization rule for these service and applet in sec.rules; 3) all services in func.rules are present on the platform (provided by some applets). We have integrated the PolicyChecker with the ClaimChecker, to ease the contract delivery. The PolicyChecker is a part of the SxCInstaller component that is the main verification interface with the Loader.

For the applet removal scenario, the PolicyChecker has to retrieve the platform policy, identify the contract of the applet to be removed and check if this applet provides any service that is listed as functionally necessary by some other applet. This is the only incompliance problem, because removal of an applet cannot introduce unauthorized service invocation. If the applet is not needed by the others, the PolicyChecker will remove its contract from the platform policy and send the new policy to the PolicyStore.

If any compliance check performed by the ClaimChecker or the PolicyChecker has failed, the components signal to the Loader, which will stop the executed change on the platform (due to the transaction mechanism on Java Card the platform will return to the previous secure state).

The PolicyStore. The PolicyStore is used to maintain the security policy across the card sessions. As the policy of the platform is dynamic in nature and cannot be static throughout the card lifecycle, it has to be stored in the EEPROM. Therefore, as we have specified in §4, the PolicyStore cannot be implemented as a part of the Native interface of the JCRE, but instead it should be the part of the Java Card interface. We have integrated the PolicyStore with the Installer component. However, the ClaimChecker needs to be integrated with the native Loader, thus the S×C framework has to be divided across the Native and the Java Card interfaces. In the same time, we need to enable the communication between these parts, in order to retrieve the current card policy and update it after changes. This communication is realized using a new dedicated Native API.

The security policy data structures were designed to be memory-saving. For example, the applet AID can occupy up to 16 bytes, therefore each called service can occupy up to 18 bytes (the interface and method tokens each occupy 1 byte). We decided to store the policy in the bit array format that allows to speed up the policy matching operations. The platform policy data structure currently supports up to 10 loaded applets, each containing up to 8 provided services; but these applets are not pre-defined and any AID can be listed in the policy.

The final implementation of the S×C framework delegates the validation for the scenario of application policy update to the PolicyStore. The reason for this decision is the fact, that it is integrated with the platform Installer, which is already exposed to the communications with the outside world. In this way we do not need to identify new protocols for the policy update, what would be necessary if we needed to invoke the PolicyChecker, and hence - the Loader. The application policy update is atomic: each time only one authorization rule can be added or removed, or one functionally necessary service can be added or removed. For the authorization service addition and functionally necessary service removal the update can be executed directly, as these changes cannot introduce inconsistency. For the removal of an authorization the PolicyStore will ensure the de-authorized client does not call this service. For the addition of a functionally necessary service the PolicyStore will check this service is actually called and is provided on the platform. If the check is successful, the update is applied, otherwise the policy is not modified.

5.3 Integration with the Java Card Platform

Fig. 8 presents the Java Card architecture extended with the S×C framework and the new steps in the applet development and deployment process. The S×C framework is fully backward-compatible with the existing Java Card platforms. The platforms that do not know about the framework will be able to process applets with contracts, because unknown Custom components are just ignored by default. The applets without the contract can be still deployed on the platform:

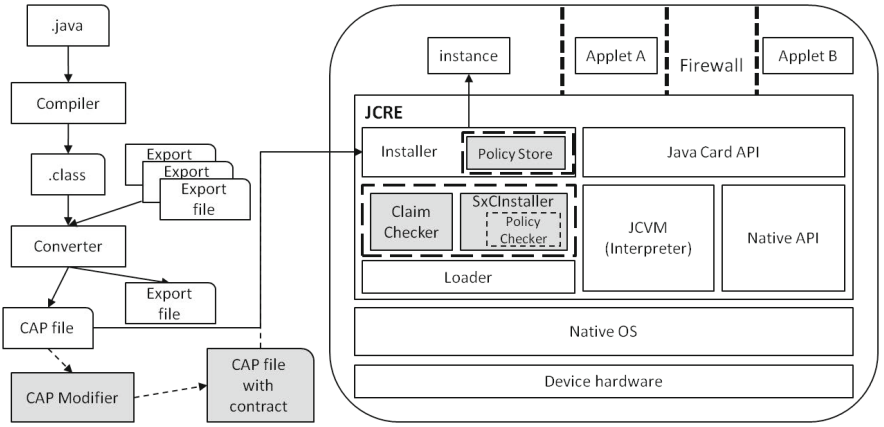


Fig. 8. The Java Card architecture and the applet deployment process in presence of the S×C framework. The grey components are introduced by the S×C approach, the dashed lines denote new steps in the development and deployment process.

they will be treated as providing and calling no services. We did not modify the standard loading protocol or the JCVM. The interested reader can find more information on the S×C design challenges and the implementation details in [7].

6 Application to the Use Case Scenarios

Let us now present concrete contracts we devised for the motivating scenarios introduced in §§2.1-2.2.

6.1 The NFC-Enabled Phone and the New Applet Installation

We consider the payment functionality of the payWave application to be implemented in the shareable `PaymentInterface` and the service `payment`. Let the Touch&Travel applet has the AID `0xAA01CA71FF10` and the payWave applet has the AID `0x4834D26C6C6F4170`[11].

We can notice in Tab. 1, which presents the contracts of the scenario applets, that one of the presented contracts of Navigo (Non-Compliant) is not compliant with the contract of the payWave application (specifically, Navigo calls the payment service, but is not authorized to do so). Therefore in our scenario, when the device holder requests the loading of Navigo with this contract, the loading will be rejected by the S×C framework. If the device holder installs the Navigo version without the non-authorized call to payWave (the Compliant option), then it will be installed without any problems.

[11] The chosen AIDs are fictional, but they are compliant with the ISO/IEC 7816-5 standard to give an idea how an actual contract can look like.

Table 1. Contracts of the payWave, Touch&Travel and Navigo applets

Contract structure	Fully-qualified names	Token identifiers
payWave		
Provides	`PaymentInterface.payment()`	⟨0, 0⟩
Calls		
sec.rules	Touch&Travel is authorized to call `PaymentInterface.payment()`	⟨ `0xAA01CA71FF10`, 0, 0⟩
func.rules		
Touch&Travel		
Provides		
Calls	payWave.`PaymentInterface.payment()`	⟨`0x4834D26C6C6F4170`, 0, 0⟩
sec.rules		
func.rules		
Navigo – Non Compliant		
Provides		
Calls	payWave.`PaymentInterface.payment()`	⟨`0x4834D26C6C6F4170`, 0, 0⟩
sec.rules		
func.rules		
Navigo – Compliant		
Provides		
Calls		
sec.rules		
func.rules		

6.2 The Telecommunications Hub and the Application Policy Update

For the application policy update scenario we present the contracts of the Consumption Optimization applet before and after the update. We consider the consumption optimzation functionality to be implemented in the shareable `OptimizationInterface` and the service `optimization`. Let the Electricity applet has the AID `0xEE06D7713386`, the Consumption Optimization applet has the AID `0xEE06D7713391`,the Gas applet has the AID `0xGG43F167B2890D6C` and the Telecare applet has the AID `0x4D357F82B1119AEE`.

Tab. 2 presents contracts the Electricity, Gas, Telecare and Consumption Optimization applets. Notice that the application policy update for addition of the authorization for Telecare is possible and will be executed by the S×C framework. The Telecare applet can later be updated and in the new version the call to the `optimization` service will appear.

7 Related Work

Fontaine et al. [5] design a mechanism for implementing transitive control flow policies on Java Card. These policies are stronger than the access control policies provided by our framework, because the S×C approach targets only direct service invocations. However, the S×C approach has the advantage of the openness of

Table 2. Contracts of the Electricity, Consumption Optimization, Gas and Telecare applets

Contract structure	Fully-qualified names	Token identifiers
Consumption Optimization – Before		
Provides	`OptimizationInterface.optmization()`	⟨0, 0⟩
Calls		
sec.rules	Electricity applet is authorized to call `OptimizationInterface.optmization()`	⟨`0xEE06D7713386`, 0, 0⟩
	Gas applet is authorized to call `OptimizationInterface.optmization()`	⟨`0xGG43F167B2890D6C`, 0, 0⟩
func.rules		
Electricity		
Provides		
Calls	ConsumptOptim.`OptimizationInterface.optimization()`	⟨`0xEE06D7713391`, 0, 0⟩
sec.rules		
func.rules		
Gas		
Provides		
Calls	ConsumptOptim.`OptimizationInterface.optimization()`	⟨`0xEE06D7713391`, 0, 0⟩
sec.rules		
func.rules		
Telecare		
Provides		
Calls		
sec.rules		
func.rules		
Consumption Optimization – After		
Provides	`OptimizationInterface.optmization()`	⟨0, 0⟩
Calls		
sec.rules	Electricity applet is authorized to call `OptimizationInterface.optmization()`	⟨`0xEE06D7713386`, 0, 0⟩
	Gas applet is authorized to call `OptimizationInterface.optmization()`	⟨`0xGG43F167B2890D6C`, 0, 0⟩
	Telecare applet is authorized to call `OptimizationInterface.optmization()`	⟨`0x4D357F82B1119AEE`, 0, 0⟩
func.rules		

the policy to any applet AID. The main limitation of [5] is the focus on ad-hoc security domains, which are very coarse grained administrative security roles (usually a handful), typically used to delegate privileges on GlobalPlatform. As a consequence we can provide a much finer access control list closer to actual practice.

An information flow verification system for small Java-based devices is proposed by Ghindici et al. [9]. The system relies on off-device and on-device steps. First, an applet *certificate* is created off device (contains information flows within the applet). Then on device the certificate is checked in a proof-carrying-code fashion and matched with the information flow policies of other applets. The information flow policies are very expressive. However, we believe the on device information flow verification for Java Card is not yet practical due to the resource and architecture limitations. The proposed system cannot be implemented for Java Card version 2.2 because the latter does not allow custom class loaders,

and even implementation for Java Card version 3.0 may not be effective due to significant amount of memory required to store the information flow policies.

There were investigations [2,3,10,11,16] of static scenarios, when all applets are known and the composition is analyzed off-device. For example, Avvenuti et al. [2] have developed the JCSI tool which verifies whether a Java Card applet set respects pre-defined information flow policies. This tool runs off-card, so it assumes an existence of a controlling authority, such as a telecom operator, that can check applets before loading.

The investigation of the Security-by-Contract techniques for Java Card is carried out in [4,8,6,7] targeting dynamic scenarios when third-party applets can be loaded on the platform.

Dragoni et al. [4] and Gadyatskaya et al. [8] propose an implementation of the PolicyChecker component as an applet. While very appealing due to avoiding the JCRE modification, it has not solved in any way the actual issue of integration with a real platform. This solution could only work if the authors of [4,8] had access to the full Java-based JCRE implementation, because only in this way the Loader can be implemented as a part of the Java Card interface. The Java Card specifications do not prohibit this, but in practice full Java-based implementations do not exist.

8 Conclusions and the Future Work

The S×C framework enables load time bytecode validation for multi-application Java cards. Now each application provider can independently deploy her applications and update the application policies. The proposed approach fits on a real smart card, it enables the backward compatibility and is not very invasive, as the changes to the platform are kept at minimum.

The main benefit of the proposed solution is the validation of the code on card. In this way each card is independent in the decision it takes; we can envisage that (U)SIM cards from the same telecom operator can contain different application sets, depending on the needs of the phone holder. The telecom operator now does not need to verify security for each possible set of applets; therefore the costs of managing the device are lower. We can envisage that the S×C approach will be quite efficient for less expensive applets (like the already mentioned messages from Santa) that do not provide and do not call any services. This fact can be easily ensured on the card itself, and these applets do not need to pass the costly certification process.

Our framework performs the load time on-card checks of Java Card bytecode. The restrictions of the Java Card platform (the dedicated service invocation instruction and the static invoked class binding in the CAP file) allow our framework to efficiently analyze the sets of invoked and provided services in the code and match them with the contract. We can notice that our bytecode analysis techniques will have to be improved, for example, following [19], before application to full Java, because the inference of method invocation targets will be more complicated. However, the idea of performing load time on-device checks

on the bytecode is promising for computationally-restricted devices, e.g. the Android phones. The users expect certain delay when an application is being installed, but they will not tolerate any runtime lags. Therefore, we think that the Security-by-Contract idea is very promising for Android and other constrained devices.

We have chosen the conservative approach for verification: if the change can lead to an insecure state it is rejected. However, this might not be acceptable in the business community. For instance, we can envisage that application providers would like to be able to revoke the access to their sensitive service at any time. The S×C framework currently does not provide this option: the access to a service for a specific client can be revoked only if this client does not actually invoke this service. To be more practical, the card should be able to perform some conflict resolution. For instance, one can choose an approach in which the application provider is always able to revoke access to her service, and the client applet will become locked until the new version without service invocation is deployed. Another possibility is to explore more the centralized policy management facilities offered by the next generations of the Java Card platform. We expect a lot of interesting research challenges in this direction.

Acknowledgements. This work was partially supported by the EU under grants EU-FP7-FET-IP-SecureChange and FP7-IST-NoE-NESSOS. We thank Eduardo Lostal for implementing the first version of the S×C prototype and Boutheina Chetali and Quang-Huy Nguyen for evaluation of the prototype and valuable information on the platform internals. We also thank Davide Sangiorgi and Elena Giachino for the invitation to give the S×C tutorial at the HATS-2012 Summer School.

References

1. Aljuraidan, J., Fragkaki, E., Bauer, L., Jia, L., Fukushima, K., Kiyomoto, S., Miyake, Y.: Run-time enforcement of information-flow properties on Android. Technical Report CMU-CyLab-12-015, Carnegie Mellon University, `http://repository.cmu.edu/cgi/viewcontent.cgi?article=1110&cgi/viewcontent.cgi?article=1110&context=cylab` (accessed on the web in December 2012)
2. Avvenuti, M., Bernardeschi, C., De Francesco, N., Masci, P.: JCSI: A tool for checking secure information flow in Java Card applications. J. of Systems and Software 85(11), 2479–2493 (2012)
3. Bieber, P., Cazin, J., Wiels, V., Zanon, G., Girard, P., Lanet, J.-L.: Checking secure interactions of smart card applets: Extended version. J. of Comp. Sec. 10(4), 369–398 (2002)
4. Dragoni, N., Lostal, E., Gadyatskaya, O., Massacci, F., Paci, F.: A load time Policy Checker for open multi-application smart cards, pp. 153–156. IEEE Press
5. Fontaine, A., Hym, S., Simplot-Ryl, I.: On-device control flow verification for Java programs. In: Erlingsson, Ú., Wieringa, R., Zannone, N. (eds.) ESSoS 2011. LNCS, vol. 6542, pp. 43–57. Springer, Heidelberg (2011)

6. Gadyatskaya, O., Lostal, E., Massacci, F.: Load time security verification. In: Jajodia, S., Mazumdar, C. (eds.) ICISS 2011. LNCS, vol. 7093, pp. 250–264. Springer, Heidelberg (2011)
7. Gadyatskaya, O., Massacci, F., Nguyen, Q.-H., Chetali, B.: Load time code validation for mobile phone Java Cards. Technical Report DISI-12-025, University of Trento (2012)
8. Gadyatskaya, O., Massacci, F., Paci, F., Stankevich, S.: Java Card architecture for autonomous yet secure evolution of smart cards applications. In: Aura, T., Järvinen, K., Nyberg, K. (eds.) NordSec 2010. LNCS, vol. 7127, pp. 187–192. Springer, Heidelberg (2012)
9. Ghindici, D., Simplot-Ryl, I.: On practical information flow policies for Java-enabled multiapplication smart cards. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 32–47. Springer, Heidelberg (2008)
10. Girard, P.: Which security policy for multiapplication smart cards? In: Proc. of USENIX Workshop on Smartcard Technology. USENIX Association (1999)
11. Huisman, M., Gurov, D., Sprenger, C., Chugunov, G.: Checking absence of illicit applet interactions: A case study. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. LNCS, vol. 2984, pp. 84–98. Springer, Heidelberg (2004)
12. Project Hydra. Project Hydra - Smart care for smart meters. Final report (October 2012), http://projecthydra.info/wp-content/uploads/2012/10/Hydra_Final_Report.pdf (accessed on the web in December 2012)
13. Koo, J., Lin, X., Bagchi, S.: PRIVATUS: Wallet-friendly privacy protection for smart meters. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 343–360. Springer, Heidelberg (2012)
14. Langer, J., Oyrer, A.: Secure element development. In: NFC Forum Spotlight for Developers, http://www.nfc-forum.org/events/oulu_spotlight/2009_09_01_Secure_Element_Programming.pdf (accessed on the web in December 2012)
15. Rial, A., Danezis, G.: Privacy-preserving smart metering. In: Proc. of the ACM WPES 2011, pp. 49–60. ACM (2011)
16. Schellhorn, G., Reif, W., Schairer, A., Karger, P., Austel, V., Toll, D.: Verification of a formal security model for multiapplicative smart cards. In: Cuppens, F., Deswarte, Y., Gollmann, D., Waidner, M. (eds.) ESORICS 2000. LNCS, vol. 1895, pp. 17–36. Springer, Heidelberg (2000)
17. Sullivan, B.: What will talking power meters say about you? http://redtape.nbcnews.com/_news/2009/10/09/6345711-what-will-talking-power-meters-say-about-you (accessed on the web in December 2012)
18. Sun. Runtime environment and virtual machine specifications. Java Card™ platform, v.2.2.2. Specification (2006)
19. Sundaresan, V., Hendren, L., Razafimahefa, C., Vallee-Rai, R., Lam, P., Gagnon, E., Godin, C.: Practical virtual method call resolution for Java. In: Proc. of OOPSLA 2000, pp. 264–280. ACM Press (2000)

Formal Aspects of Free and Open Source Software Components*

A Short Survey

Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli

Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126, CNRS
roberto@dicosmo.org, {treinen,zack}@pps.univ-paris-diderot.fr

Abstract. Free and Open Source Software (FOSS) distributions are popular solutions to deploy and maintain software on server, desktop, and mobile computing equipment. The typical deployment method in the FOSS setting relies on software *distributions* as vendors, *packages* as independently deployable components, and *package managers* as upgrade tools. We review research results from the past decade that apply formal methods to the study of inter-component relationships in the FOSS context. We discuss how those results are being used to attack both issues faced by users, such as dealing with upgrade failures on target machines, and issues important to distributions such as quality assurance processes for repositories containing tens of thousands, rapidly evolving software packages.

1 Introduction

Free and Open Source Software [47], or FOSS, is used daily, world-wide, to manage computing infrastructures ranging from the very small, with embedded devices like Android-based smart phones, to the very big, with Web servers where FOSS-based solutions dominate the market. From the outset, most FOSS-based solutions are installed, deployed, and maintained relying on so-called *distributions*. The aspect of software distributions that will interest us in this paper is that they provide a *repository*: a typically large set of software *packages* maintained as software components that are designed to work well together. Software distributions, like for instance GNU/Linux distributions, have in fact additional aspects that are crucial but which are not considered in this work, like for instance an installer that allows a user to install an initial system on a blank machine, or infrastructure for interaction between users and developers like a bug tracking system.

While specific technologies vary from distribution to distribution, many aspects, problems, and solutions are common across distributions. For instance, packages have expectations on the deployment context: they may require other

* Work partially supported by Aeolus project, ANR-2010-SEGI-013-01, and performed at IRILL, center for Free Software Research and Innovation in Paris, France, www.irill.org

E. Giachino et al. (Eds.): FMCO 2012, LNCS 7866, pp. 216–239, 2013.

packages to function properly—declaring this fact by means of *dependencies*—and may be incompatible with some other packages—declaring this fact by means of *conflicts*. Dependencies and conflicts are captured as part of *package metadata*. Here as an example showing the popular Firefox web browser as a package in the Debian distribution:

```
Package: firefox
Version: 18.0.1-1
Depends: libc6 (>= 2.4), libgtk2.0-0 (>= 2.10), libstdc++6,
  fontconfig, procps, xulrunner-18.0, libsqlite3-0, ...
Suggests: fonts-stix | otf-stix, mozplugger,
  libgssapi-krb5-2 | libkrb53
Conflicts: mozilla-firefox (<< 1.5-1)
Provides: www-browser, gnome-www-browser
```

A couple of observations are in order. First, note how the general form of inter-package relationships (conflicts, dependencies, etc.) is that of propositional logic formulae, having as atoms predicates on package names and their versions. Second, we have various degrees of dependencies, strong ones (like "Depends") that must be satisfied as deployment preconditions and weak ones (like "Suggests" and "Recommends", the latter not shown). Finally, we also observe an indirection layer in the package namespace implemented by "Provides". Provided packages are sometimes referred to as *features*, or *virtual packages* and mean that the providing package can be used to satisfy dependencies for—or induce conflicts with—the provided name.

To maintain package assemblies, semi-automatic *package manager* applications are used to perform package installation, removal, and upgrades on target machines—the term *upgrade* is often used to refer to any combination of those actions. Package managers incorporate numerous functionalities: trusted retrieval of components from remote repositories, planning of upgrade paths in fulfillment of deployment expectations (also known as *dependency solving*), user interaction to allow for interactive tuning of upgrade plans, and the actual deployment of upgrades by removing and adding components in the right order, aborting the operation if problems are encountered at deploy-time [24].

Unfortunately, due to the sheer size of package repositories in popular FOSS distributions (in the order of tens of thousands [33]), several challenges need to be addressed to make the distribution model viable in the long run. In the following we will focus on two classes of issues and the related research directions:

1. issues faced by distribution *users*, who are in charge of maintaining their own installations, and
2. issues faced by *distribution editors*, who are in charge of maintaining the consistency of distribution repositories.

As motivating example of issues that are faced by users consider the seemingly simple requirement that a package manager should change as little as possible on the target machine in order to satisfy user requests. Unfortunately, as demonstrated in Fig. 1, that property is not yet offered by most mainstream package

```
# aptitude install baobab
[...]
The following packages are BROKEN: gnome-utils
The following NEW packages will be installed: baobab [...]
The following actions will resolve these dependencies:
Remove the following packages: gnome gnome-desktop-environment libgdict-1.0-6
Install the following packages: libgnome-desktop-2 [2.22.3-2 (stable)]
Downgrade the following packages:
  gnome-utils [2.26.0-1 (now) -> 2.14.0-5 (oldstable)] [...]
0 packages upgraded, 2 newly installed, 1 downgraded,
180 to remove and 2125 not upgraded. Need to get 2442kB
of archives. After unpacking 536MB will be freed.
Do you want to continue? [Y/n/?]
```

Fig. 1. Attempt to install a disk space monitoring utility (called *baobab*) using the Aptitude package manager. In response to the request, the package manager proposes to downgrade the GNOME desktop environment all together to a very old version compared to what is currently installed. As shown in [4] a trivial alternative solution exists that minimizes system changes: remove a couple of dummy "meta" packages.

managers. A related issue, that we will also discuss in the following, is that of providing expressive languages that allow users of package managers to express their preferences, e.g. the demand to minimize the size occupied by packages installed on their machines.

Distribution editors, on the other hand, face the challenging task of avoiding inconsistencies in huge package archives. A paradigmatic example of inconsistency that they should avoid is that of shipping *uninstallable packages*, i.e. packages that, no matter what, cannot be installed on user machines because there is no way to satisfy their dependencies and conflicts. Consider for instance the following (real) example involving the popular Cyrus mail system:

Package: `cyrus-common-2.2`	**Package**: `cyrus-common-2.4`
Version: `2.4.12-1`	**Version**: `2.4.12-1`
Depends: `cyrus-common-2.4`	**Conflicts**: `cyrus-common-2.2`

It is easy to verify that it is not possible to install the above `cyrus-common-2.2` package—a dummy package made to ease upgrades to Cyrus 2.4—out of any package repository that also contains the `cyrus-common-2.4` package shown in the example. Even worse, it can be shown that the issue is not transitional, i.e. the team responsible for `cyrus-common-2.2` (its *maintainers*) cannot simply wait for the issue to go away, they have to manually fix the metadata of their package so that the cause of the uninstallability goes away. The challenge here is that, while it is easy to reason on simple cases like this one, distribution editors actually need semi- or fully-automated tools able to spot this kind of quality assurance issues and point them to the most likely causes of troubles.

Paper Structure. In the following we provide a short summary of research from the past decade on the formal aspects of FOSS packages. We first present, in Sect. 2, different formal models able to capture the parts of package metadata

that are relevant to attack both issues faced by users and by distributions. Then, in Sect. 3, we give an overview of results that foster the development of complete and expressive package managers that would provide a better package management experience to users. Finally, in Sect. 4, we do the same with research results that have been used to develop and deploy semi-automated quality assurance tools used daily by editors of popular FOSS distributions to assess the quality of their package repositories.

2 Formal Package Models

Different formal treatments of packages and their relationships are needed for different purposes. Two main approaches have been devised: a syntactic (or *concrete*) one which captures the syntax of inter-package relationships, so that they can be treated symbolically, similarly to how package maintainers reason about them. We will use such an approach to reason about the future evolution of repositories (see Sect. 4), taking into account yet unknown package versions.

A more abstract package model is useful too, in order to make the modeling more independent from specific component technologies and their requirement languages. We will use this kind of modeling to recast the problem of verifying package installability as a SAT problem (see Sect. 2.3).

2.1 Concrete Package Model

A concrete package model, originally inspired by Debian packages, has been given in [5] and further detailed in [6]. In this model packages are captured as follows:

Definition 1 (Package). *A* package (n, v, D, C) *consists of*

- *a* package name $n \in \mathrm{N}$,
- *a* version $v \in \mathrm{V}$,
- *a set of dependencies* $D \subseteq \wp(\mathrm{N} \times \textsc{Con})$,
- *a set of conflicts* $C \subseteq \mathrm{N} \times \textsc{Con}$,

where N is a given set of possible package names, V a set of package versions, and Con a set of syntactic constraints on them like $\top$, $= v$, $> v$, $\leq v$, ... The intuition is that dependencies should be read as *conjunctions of disjunctions*. For example: $\{\{(p, \geq 1), (q, = 2)\}, \{(r, < 5)\}\}$ should be read as $((p \geq 1) \vee (q = 2)) \wedge (r < 5)$. Starting from this intuition, the expected semantics of package constraints can be easily formalized.

Notation 1. *Given a package p we write p.n (resp. p.v, p.D, p.C) for its name (resp. version, dependencies, conflicts).*

Repositories can then be defined as package sets, with the additional constraint that name/version pairs are unambiguous package identifiers:

Table 1. Sample package repository

```
Package: a                Package:  b             Package: d
Version: 1                Version:  2             Version: 3
Depends: b (≥ 2) | d      Conflicts: d

Package: a                Package:  c             Package: d
Version: 2                Version:  3             Version: 5
Depends: c (> 1)          Depends:  d (> 3)
                          Conflicts: d (= 5)
```

Definition 2 (Repository). *A* repository *is a set of packages, such that no two different packages carry the same name and version.*

A pair of a name and a constraint has a meaning with respect to a given repository R, the precise definition of which would depend on the formal definition of constraints and their semantics:

Notation 2. *Given a repository R, $n \in \mathrm{N}$ and $c \in \textsc{Con}$, we write $[[(n, c)]]_R$ for the set of packages in R with name n and whose version satisfies the constraint c.*

We can then finally capture the important notions of installation and of (co-)installability:[1]

Definition 3 (Installation). *Let R be a repository. An R-installation is a set of packages $I \subseteq R$ such that $\forall p, q \in I$:*

abundance *for each element $d \in p.D$ there exists $(n, c) \in d$ and a package $q \in I$ such that $q \in [[(n, c)]]_R$.*
peace *for each $(n, c) \in p.C$: $I \cap [[(n, c)]]_R = \emptyset$*
flatness *if $p \neq q$ then $p.n \neq q.n$*

Definition 4 (Installability). *$p \in R$ is R-installable if there exists an R-installation I with $p \in I$.*

Definition 5 (Co-Installability). *$S \subseteq R$ is R-co-installable if there exists an R-installation I with $S \subseteq I$.*

Example 1 (Package Installations). Consider the repository R shown in Table 1. The following sets are not R-installations:

– R as a whole, since it is not flat;

[1] We remind that this is a *specific* concrete package model, inspired by Debian packages. Therefore not all installation requirements listed here have equivalents in *all* component technologies. Most notably the presence of the flatness property varies significantly from technology to technology. As discussed in [4,6] this does not affect subsequent results.

– $\{(a,1),(c,3)\}$, since both a's and b's dependencies are not satisfied;
– $\{(a,2),(c,3),(d,5)\}$, since there is a conflict between c and d.

The following sets are valid R-installations: $\{(a,1),(b,2)\}$, $\{(a,1),(d,5)\}$. We can therefore observe that the package $(a,1)$ is R-installable, because it is contained in an R-installation.

The package $(a,2)$ is not R-installable because any installation of it must also contain $(c,3)$ and consequently $(d,5)$, which will necessarily break peace. □

2.2 Abstract Package Model

A more abstract package model [43] was the basis for several of the studies discussed in the present work. The key idea is to model repositories as non mutable entities, under a closed world assumption stating that we know the set of all existing packages, that is that we are working with respect to a given repository R.

Definition 6. *An* abstract repository *consists of*

– *a set of packages* P,
– *an anti-reflexive and symmetric* conflict relation $C \subseteq P \times P$,
– *a* dependency function $D\colon P \longrightarrow \wp(\wp(P))$.

The nice properties of peace, abundance, and (co-)installability can be easily recast in such a model.

The concrete and abstract models can be related. In particular, we can translate instances of the concrete model (easily built from real-life package repositories) into instances of the more abstract model, preserving the installability properties. To do that, the main intuition is that (concrete) package constraints can be "expanded" to disjunctions of all (abstract) packages that satisfy them. For example, if we have a package p in versions 1, 2, and 3, then a dependency on $p \geq 2$ will become $\{\{(p,2),(p,3)\}\}$. For conflicts, we will add a conflict in the abstract model when either one of the two (concrete) packages declare a conflict on the other, or when we have two packages of the same name and different versions. The latter case implements the flatness condition. Formally:

Notation 3. *Let R be a repository in the concrete model. We can extend the semantics of pairs of names and constraints to sets as follows:*

$$[[\{(n_1,c_1),\ldots,(n_m,c_m)\}]]_R = [[(n_1,c_1)]]_R \cup \ldots \cup [[(n_m,c_m)]]_R$$

Definition 7 (Concrete to Abstract Model Translation). *Let R be a repository in the concrete model. We define an abstract model $R_a = (P_a, D_a, C_a)$.*

– *P_a: the same packages as in R*
– *We define the dependency in the abstract model:*

$$D_a(p) = \{[[\phi]]_R \mid \phi \in p.D\}$$

– *We define conflicts in the abstract model:*

$$\begin{aligned} C_a = \{&(p_1,p_2) \mid p_1 \in [[p_2.C]]_R \vee p_2 \in [[p_1.C]]_R\} \\ \cup\, \{&(p_1,p_2) \mid p_1.n = p_2.n \wedge p_1.v \neq p_2.v\} \end{aligned}$$

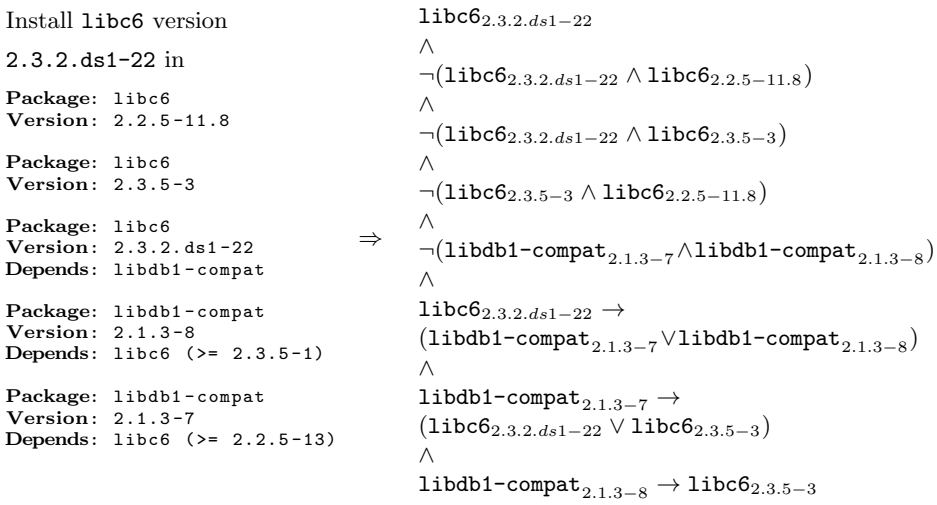


Fig. 2. Example: package installability as SAT instance

2.3 On the Complexity of Installability

Now that we have rigorously established the notion of package (co-)installability, it is legitimate to wonder about the complexity of deciding these properties. Is it "easy enough" to automatically identify non-installable packages in large repositories of hundreds of thousands of packages? The main complexity result, originally established in [43], is not encouraging:

Theorem 1. *(Co-)installability is NP-hard (in the abstract model).*

The gist of the proof is a bidirectional mapping between boolean satisfiability (SAT) [19] and package installability. For the forward mapping, from packages to SAT, we use one boolean variable per package (the variable will be true if and only if the corresponding package is installed), we expand dependencies as implications $p \rightarrow (r_1 \vee \cdots \vee r_n)$ where r_i are all the packages satisfying the version constraints, and encode conflicts as $\neg(p \wedge q)$ clauses for every conflicting pair (p, q). Thanks to this mapping, we can use SAT solvers for checking the installability of packages (see Fig. 2 and Sect. 4).

The backward mapping, from SAT to package installability, can be established considering 3-SAT instances, as detailed in [30].

Given that the proof is given for the abstract model, one might wonder to which kind of concrete models it applies. The question is particularly relevant to know whether dependency solving in the context of specific component technologies can result in corner cases of unmanageable complexity or not. Several instances of this question have been answered in [4], considering the common features of several component models such as Debian and RPM packages, OSGi [45] bundles, and Eclipse plugins [20,15]. Here are some general results:

- Installability is NP-complete provided the component model features conflicts and disjunctive dependencies.
- Installability is in PTIME if the component model does *not* allow for conflicts (neither explicitly, nor implicitly with clauses like Eclipse's "singleton").
- Installability is in PTIME if the component model does not allow for disjunctive dependencies or features, and the repository does not contain multiple versions of packages.

3 Upgrade Optimization

The discussed complexity results provide convincing evidence that dependency solving is difficult to get right, more than developers might imagine at first. Several authors [39,34,40,55,51,54,24,36] have pointed out two main deficiencies of state-of-the-art package managers in the area of dependency solving—incompleteness and poor expressivity—some of them have proposed various alternative solutions.

A *dependency solving problem*, as usually faced by dependency solvers, can be described as consisting of: *(i)* a repository of all available packages (sometimes also referred to as a *package universe*); *(ii)* a subset of it denoting the set of currently installed packages on the target machine (*package status*); *(iii)* a *user request* usually asking to install, upgrade, or remove some packages. The expected output is a new package status that both is a proper installation (in the sense of Def. 3) and satisfies the user request. Note that, due to the presence of both implicit and explicit disjunctions in the dependency language, there are usually many valid solutions for a given upgrade problem. In fact, it has been shown in [4] that there are *exponentially* many solutions to upgrade problems in all non-trivial repositories.

A dependency solver is said to be *complete* if it is able to find a solution to an upgrade problem whenever one exists.

Given the huge amount of valid solutions to any given upgrade problem, we need languages that allow the user to express her preferences such as "favor solutions that minimize the amount of used disk space", "favor solutions that minimize the changes to the current package status", "do not install packages that are affected by outstanding security issues", etc.

Unfortunately, most state-of-the-art package managers are neither complete nor offer expressive user preference languages [53].

3.1 The Common Upgradeability Description Format

CUDF [52,4] (the Common Upgradeability Description Format)[2] is a language devised to solve the issues of completeness and expressivity by inducing a synergy among package managers developers and researchers in the various fields of constraint solving. At first glance, a CUDF document captures an instance of a

[2] `http://www.mancoosi.org/cudf/`, retrieved May 2013

```
preamble:
property: bugs: int = 0, suite: enum(stable,unstable) = "stable",

package: car
version: 1
depends: engine, wheel > 2, door, battery <= 13
installed: true
bugs: 183

package: bicycle
version: 7
suite: unstable

package: gasoline-engine
version: 1
depends: turbo
provides: engine
conflicts: engine, gasoline-engine
installed: true
...
request:
install: bicycle, gasoline-engine = 1
upgrade: door, wheel > 3
```

Fig. 3. Sample CUDF document

dependency solving problem using a human readable syntax, as shown in Fig. 3. CUDF is an extensible language—i.e. it allows to represent ad-hoc package properties that can then be used to express user preferences—and provides a formal semantics to unambiguously determine whether a given solution is correct with respect to the original upgrade problem or not.

CUDF is also neutral on both specific packaging and solving technologies. Several kinds of package manager-specific upgrade problems can be encoded in CUDF and then fed to solvers based on different constraint solving techniques. Fig. 4 enumerates a number of packaging technologies and solving techniques that can be used together, relying on CUDF for data exchange.

This is achieved by instrumenting existing package managers with the ability to communicate via the CUDF format with external dependency solvers. Such an arrangement, depicted in Fig. 5 and studied in [3,7], allows to share dependency solvers across package managers. Several modern package managers have followed this approach either offering the possibility to use external CUDF solvers as plugins, or even abandoning the idea of an integrated solver and always using external CUDF solvers. Examples are the APT package manager used by the Debian and Ubuntu distributions, the P2 provisioning platform for Eclipse plugins, and the OPAM package manager for the OCaml language.

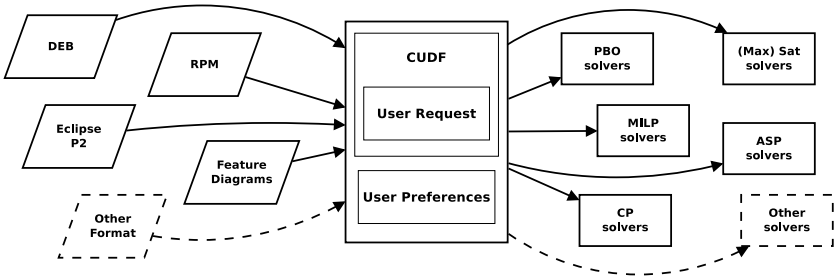


Fig. 4. Sharing upgrade problems and solvers among communities

3.2 User Preferences

In itself, CUDF does not mandate a specific language for user preferences, but *supports* them, in various ways. On one hand, CUDF captures and exposes all relevant characteristics of upgrade problems (e.g. package and user request properties) that are needed to capture user preferences in common scenarios [53]. Also, CUDF does so in an extensible way, so that properties that are specific to a given package technology can still be captured. On the other hand, the CUDF model is rigorous, providing a solid base to give a clear and measurable semantics to user preferences, which would allow to compare solutions and decide how well they score w.r.t. user preferences.

Several proposals of user preference languages have been advanced. The main challenge consists in finding a middle ground between the expressivity that users desire and the capabilities of modern constraint solvers.

Historically, OPIUM [55] has used SAT-based optimization to hard-code a fairly typical user preference, corresponding to the desire of minimizing the number of packages that are installed/removed to satisfy user request.

For the first time in [4], a flexible preference language has been proposed, based on a set of metrics that measure the distance between the original package status and the solution found by the dependency solver. Distance can be measured on various axes: the number of packages removed, newly installed, changed, that are not up to date (i.e. not at the latest available version), and with unsatisfied "weak" dependencies (i.e. packages that are "recommended" to be installed together with others, but not strictly required). Those metrics can be combined using a dictionary of aggregation functions that are commonly supported by solvers capable of multicriteria optimization [49], in particular lexicographic orderings and weighted sums. Using the resulting formalism it is possible to capture common user preference use cases such as the following *paranoid* one

$$paranoid = lex(-removed, -changed)$$

The solution scoring best under this criterion is the one with the smallest number of removed functionalities, and then with the smallest number of changes

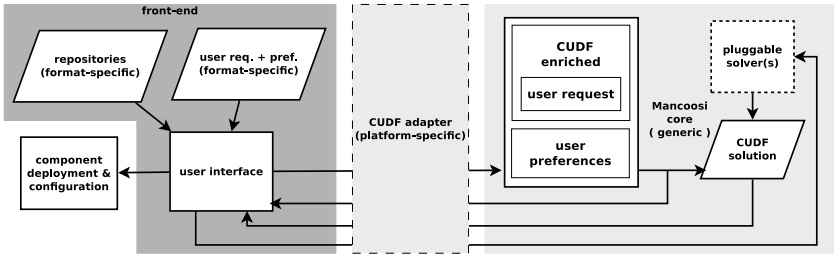


Fig. 5. Modular package manager architecture

(e.g. upgrade/downgrade/install actions). A *trendy* preference, i.e. the desire of having the most recent versions of packages, is also easy to write as:

$$trendy = lex(-removed, -notuptodate, -unsatrec, -new)$$

This set of preference combinators is bound to grow to encompass new user needs. For example, it is often the case that a single source package can produce many binary packages, and that using a mix of binary packages coming from different versions of the same source package is problematic. In recent work, it has been shown how to implement an optimization criterion that allows to specify that some packages need to be *aligned*, for different notions of alignment [23].

3.3 The MISC Competition

The existence of a language like CUDF allows to assemble a corpus of challenging (for existing dependency solvers) upgrade problems coming from actual users of different package managers. Using such a corpus, researchers have established the MISC (for Mancoosi International Solver Competition)[3] that has been run yearly since 2010. The goal of the competition is to advance the state of the art of real dependency solvers, similarly to what has happened in others fields with, e.g., the SAT competition [35].

A dozen solvers have participated in the various editions, attacking CUDF-encoded upgrade problems using solvers based on a wide range of constraint solving techniques. Table 2 shows a sample of MISC participants from the 2010 and 2011 editions.

Analysis of the competition results has allowed us to experimentally establish the limits of state-of-the-art solvers. In particular, they have been shown to significantly degrade their ability to (quickly) find solutions as the number of used package repositories grows, which is a fairly common use case. Each competition edition has established one or more winners, e.g. one winner in the *trendy* track and one in the *paranoid* one. Modular package managers that follow the architecture of Fig. 5 can then use winning solvers, or other entrants, as their dependency solver of choice.

[3] `http://www.mancoosi.org/misc/`, retrieved May 2013

Table 2. Sample of MISC competition entrants, ed. 2010 and 2011

solver	author/affiliation	technique/solver
apt-pbo [54]	Trezentos / Caixa Magica	Pseudo Boolean Optimization
aspcud	Matheis / University of Potsdam	Answer Set Programming
inesc [9]	Lynce et. al / INESC-ID	Max-SAT
p2cudf [9]	Le Berre and Rapicault / Univ. Artois	Pseudo Boolean Optimization / Sat4j (`www.sat4j.org`)
ucl	Gutierrez et al. / Univ. Luvain	Graph constraints
unsa [44]	Michel et. al / Univ. Sophia-Antipolis	Mixed Integer Linear Programming / CPLEX (`www.cplex.com`)

Solvers able to handle these optimization combinators can also be used for a variety of other purposes. It is worth mentioning one of the most unusual, which is building minimum footprint virtual images for the cloud: as noticed in [46], virtual machine images often contain largely redundant package selections, wasting disk space in cloud infrastructures. Using the toolchain available in the `dose` library,[4] which is at the core of the MISC competition infrastructure, one can compute the smallest distribution containing a given set of packages. This problem has actually been used as one of the track of the 2012 edition of the MISC competition.

More details on CUDF and MISC are discussed in [4,7].

4 Quality Assurance of Component Repositories

A particularly fruitful research line tries to solve the problems faced by the maintainers of component repositories, and in particular of FOSS distributions.

A distribution maintainer controls the evolution of a distribution by regulating the flow of new packages into and the removal of packages from it. With the package count in the tens of thousands (over 35.000 in the latest Debian development branch as of this writing), there is a serious need for tools that help answering *efficiently* several different questions. Some are related to the current state of a distribution, like: "What are the packages that cannot be installed (i.e., that are *broken*) using the distribution I am releasing?", "what are the packages that block the installation of many other packages?", "what are the packages most depended upon?". Other questions concern more the evolution of a distribution over time, like: "what are the *broken* packages that can only be fixed by changing them (as opposed to packages they depend on)?", "what are the *future version changes* that will break the most packages in the distribution?", "are there sets of packages that were installable together in the previous release, and can no longer be installed together in the next one?".

In this section, we highlight the most significant results obtained over the past years that allow to answer some of these questions, and led to the development of tools which currently are being adopted by distribution maintainers.

[4] `http://www.mancoosi.org/software/`, retrieved May 2013

4.1 Identifying Broken Packages

As we have seen in Sect. 2.3, the problem of determining whether a single package is installable using packages from a given repository is NP-hard. Despite this limiting result, modern SAT solvers are able to handle easily the instances coming from real world repositories. This can be explained by observing that explicit conflicts between packages are not very frequent, even if they are crucial when they exist, and that when checking installability in a single repository one usually finds only one version per package, hence no implicit conflicts. As a consequence, there is now a series of tools, all based on the original `edos-debcheck` tool developed by Jérôme Vouillon in 2006 [43], that are part of the `dose` library and can check installability of Debian or RPM packages, as well as Eclipse plugins, in a very short time: a few seconds on commodity desktop hardware are enough to handle the ≈ 35.000 packages from the latest Debian distribution.[5]

4.2 Analyzing the Dependency Structure of a Repository

Identifying the packages that are not installable in a repository is only the first basic analysis which is of interest for a distribution maintainer: among the large majority of packages that are installable, not all have the same importance, and not all can be installed together.

It is quite tempting, for example, to use the number of incoming dependencies on a package as a measure of its importance. Similarly, it is tempting to analyze the dependency graph trying to identify "weak points" in it, along the tradition of studies on small-world networks [8]. Several studies in the literature do use explicit dependencies, or their transitive closure, to similar ends (e.g. [37,42]). The explicit, syntactic dependency relation $p \rightarrow q$ is however too imprecise for them and can be misleading in many circumstances. Intuitively, this is so because paths in the explicit dependency graph might connect packages that are incompatible, in the sense that they cannot be installed together. To solve that problem we need to distinguish between the syntactic dependency graph and a more meaningful version of it that takes into account the actual semantics of dependencies and conflicts.

This was the motivation for introducing the notion of *strong dependency* [1] to identify the packages that are at the core of a distribution.

Definition 8. *A package p strongly depends on q (written $p \Rightarrow q$) with respect to a repository R if it is not possible to install p without also installing q.*

This property is easily seen equivalent to the implication $p \rightarrow q$ in the logical theory obtained by encoding the repository R, so in the general case this problem is co-NP-complete, as it is the dual of an installation problem, and the strong dependency graph can be huge, because it is transitive. Nevertheless, it is possible on practical instances to compute the strong dependency graph of a recent

[5] A daily updated showcase of uninstallable Debian packages, used by the distribution for quality assurance purposes, is currently available at `http://edos.debian.net/edos-debcheck/` (retrieved January 2013).

Table 3. Top sensitive packages in Debian 5.0 "Lenny"

#	package	deps	**s. deps**	closure
1	`gcc-4.3-base`	43	**20128**	20132
2	`libgcc1`	3011	**20126**	20130
3	`libc6`	10442	**20126**	20130
4	`libstdc++6`	2786	**14964**	15259
5	`libselinux1`	50	**14121**	14634
6	`lzma`	4	**13534**	13990
7	`libattr1`	110	**13489**	14024
8	`libacl1`	113	**13467**	14003
9	`coreutils`	17	**13454**	13991
10	`dpkg`	55	**13450**	13987
11	`perl-base`	299	**13310**	13959
12	`debconf`	1512	**11387**	12083
13	`libncurses5`	572	**11017**	13466
14	`zlib1g`	1640	**10945**	13734
15	`libdb4.6`	103	**9640**	13991
	...			

Debian distribution in a few hours on a modern multicore machine. The optimized algorithms able to do so have been discussed in [1] and are implemented in the `dose` toolchain.[6]

Once the strong dependencies graph is known, it is possible to define the *impact set* of a package, as the set of packages that strongly depend on it: this is a notion of robustness, as removing p from the distribution renders uninstallable all packages in its impact set, while this is not the case if one uses direct or transitive dependencies.

In Table 3 are shown the ten packages from the Debian Lenny distribution with the largest impact set, and it is easy to see that the number of direct incoming dependencies is totally unrelated to the real importance of the package, while the number of transitive incoming dependencies is always an overapproximation.

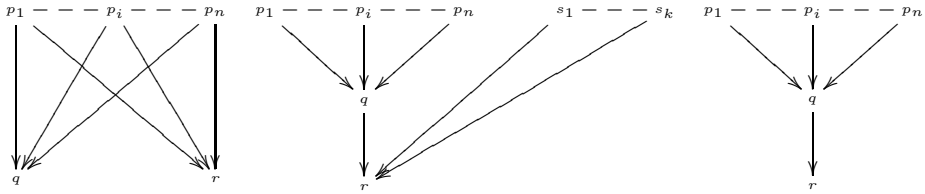
In the list of Table 3, a knowledgable maintainer will recognize a cluster of interrelated packages: `gcc-4.3-base`, `libgcc1` and `libc6` are all essential components of the C library, and they have similar sized impact sets. In the general case, though, as shown by the examples configurations drawn in Fig. 6, two packages having similar sized impact sets need not be correlated.

To identify those packages that are correlated, and identify the most relevant ones among them, one can define, on top of the strong dependency graph, a dominance relation similar to the one used in traditional flow graphs [41].

Definition 9 (Strong Dominance). *We say that q strongly dominates r if:*

- *q strongly depends on r, and*
- *every package that strongly depends on r also strongly depends on q.*

[6] `http://www.mancoosi.org/software/`, retrieved May 2013



(a) Coincidence (b) General case (c) Dominance

Fig. 6. Significant configurations in the strong dependency graph

Intuitively, in a strong dominance configuration, that looks like Fig. 6c, the strong dependency on r of packages in its impact set is "explained by" their strong dependency on q.

Strong dominance can be computed efficiently [13] and properly identifies many relevant clusters of packages related by strong dependencies like the `libc6` one, but the tools built on the work of [1] also allow to capture partial dominance situations as in Fig. 6b.

4.3 Analyzing the Conflict Structure of a Repository

In a repository there are packages that can be installed in isolation, but not together: despite the existence of interesting approaches that make it possible to reduce the need for package conflicts when installing user-level packages [29], there are good reasons for making sure, for example, that only one mail transport agent, or database server, is installed on a given machine, and that only one copy of a dynamic library needs to be updated if a security issue is uncovered. This is why in the current Debian distribution one can find over one thousand explicit conflicts declared among packages [10,11].

Once conflicts are part of a distribution, it is important to be able to assess their impact, identifying those packages that are incompatible. This is not an easy task, even when looking just for incompatibilities between package pairs, that are known as *strong conflicts* [17], as opposed to "ordinary" conflicts.

Definition 10 (Strong Conflicts). *The packages in S are in* strong conflict *if they can never be installed all together.*

Indeed, by duality with the installability problem, one obtains the following

Theorem 2. *Determining whether S is in strong conflict in a repository R is co-NP-complete.*

In practice, though, strong conflicts can be computed quite efficiently [25], and this allows to identify packages with an abnormal number of incompatibilities, that are simply undetectable using other kinds of metrics. For example, Table 4 lists the ten packages from Debian Lenny with the highest number of strong

conflicts, and the package `ppmtofb` clearly stands out as a problematic one: it is installable, so it will not be flagged by the `edos-debcheck` tool, but it is in practice incompatible with a large part of the Debian distribution. It turned out that this package depended on an old version of Python, which was phased out, but was never updated; after reporting the issue, the package was dropped from the distribution, because of lack of maintainers, but could have been adapted to the latest Python versions too.

Table 4. Top packages with the highest number of strong conflicts in Debian Lenny

Strong Conflicts	Package	Explicit Conflicts	Explicit Dependencies	Cone Size	Cone Height
2368	ppmtofb	2	3	6	4
127	libgd2-noxpm	4	6	8	4
127	libgd2-noxpm-dev	2	5	15	5
107	heimdal-dev	2	8	121	10
71	dtc-postfix-courier	2	22	348	8
71	dtc-toaster	0	11	429	9
70	citadel-mta	1	6	123	9
69	citadel-suite	0	5	133	9
66	xmail	4	6	105	8
63	apache2-mpm-event	2	5	122	10

More generally, one is interested in identifying the sets of packages that are incompatible, and in providing a way for a distribution maintainer to visualize the problematic configurations. With over 35.000 packages, and hundreds of thousands of relationships, this may look like an impossible task.

The key idea for properly addressing this challenge is to extract from the original, huge repository, a much smaller one, called a *coinstallability kernel*, that contains a representative of each package of the original repository, and preserves co-installability of packages [25]. That is, even if a coinstallability kernel is a much more compact representation of package relationships than the original one, all relevant information to decide whether packages are co-installable or not is retained by it.

To obtain a coinstallability kernel, we start from the original repository and perform a series of transformations on it. As a first step, one builds a transitive closure of the dependency relation, reminiscent of the conversion to conjunctive normal form of propositional formulae, but dropping at the same time some redundant dependencies to avoid combinatorial explosion. This phase produces a repository that has a two-level structure, which one may simplify further by removing other redundant dependencies that are exposed by the transitivization phase; after closing the dependency function reflexively, one can finally collapse packages that have the same behavior with respect to co-installability into equivalence classes, and then remove the reflexive dependencies to obtain a compact visualization of the kernel of the repository.

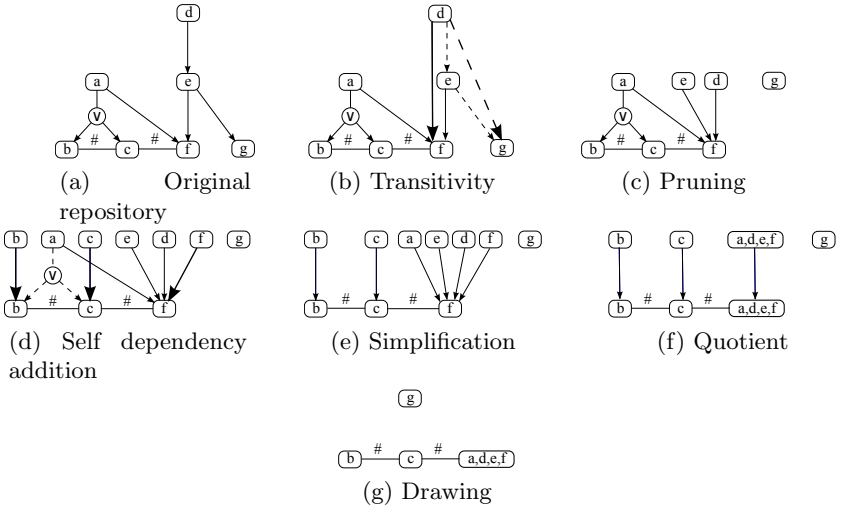


Fig. 7. Transformations of a repository. Conflicts edges are denoted with #; arrows denote direct (conjunctive) dependencies, whereas disjunctive dependencies use explicit "OR" nodes. Dashed lines are used, at each phase, to highlight edges that will disappear in the next phase.

These phases are shown on a sample repository in Fig. 7: it is clear from this example that in the final repository package `g` can always be installed, while `b` and `c` are incompatible, and all the packages `a, d, e, f` behave the same with respect to co-installability, and are only incompatible with `c`. Due to the fundamental theorems proved, and machine checked, in [25], this is also the case in the original repository.

On real-world repositories, the simplification obtained is astonishing, as can be seen in the following table, that also indicates the running time of the `coinst` tool[7] on a commodity laptop:

	Debian		Ubuntu		Mandriva	
	before	after	before	after	before	after
Packages	28919	1038	7277	100	7601	84
Dependencies	124246	619	31069	29	38599	8
Conflicts	1146	985	82	60	78	62
Running time (s)		10.6		1.19		11.6

4.4 Predicting Evolutions

Some aspects of the quality assessment in FOSS distributions are best modeled by using the notion of *futures* [2,5] of a package repository. This allows to investigate under which conditions a potential future problem may occur, or what

[7] `http://coinst.irill.org`, retrieved January 2013

```
Package: bar                    Package: foo
Version: 2.3                    Version: 1
                                Depends: (baz (=2.5) | bar (=2.3)),
Package: baz                             (baz (<2.3) | bar (>2.6))
Version: 2.5
Conflicts: bar (> 2.4)
```

Fig. 8. Package `foo` in version 1 is outdated

changes to a repository are necessary to make a currently occurring problem go away. This analysis can give package maintainers important hints about how problems may be solved, or how future problems may be avoided. The precise definition of these properties relies on the definition of the possible future of a repository:

Definition 11 (Future). *A repository F is a* future *of a repository R if the following two properties hold:*

uniqueness *$R \cup F$ is a repository; this ensures that if F contains a package p with same version and name as a package q already present in R, then $p = q$;*
monotonicity *For all $p \in R$ and $q \in F$: if $p.n = q.n$ then $p.v \leq q.v$.*

In other words, when going from the current repository to some future of it one may upgrade current versions of packages to newer versions, but not downgrade them to older versions (monotonicity). One is not allowed to change the meta-data of a package without increasing its version number (uniqueness), but besides this the upgrade may modify the meta-data of a package in any possible way, and may even remove a package completely from the repository, or introduce new packages. This notion models all the changes that are possible in the maintenance process usually used by distribution editors, even if the extreme case of a complete change of meta-data allowed in this model is quite rare in practice. Note that the notion of future is not transitive as one might remove a package and then reintroduce it later with a lower version number.

The first property using futures that we are interested in is the following one:

Definition 12 (Outdated). *Let R be a repository. A package $p \in R$ is* outdated *in R if p is not installable in any future F of R.*

That is, p is outdated in R if it is not installable (since R is itself one of its futures) and if it has to be upgraded to make it ever installable again. In other words, the only way to make p installable is to upload a fixed version of the package since no modification to other packages than p can make p installable. This information is useful for quality assurance since it pinpoints packages where action is required. An example of an outdated package is given in Fig. 8.

Definition 13 (Challenges). *Let R be a repository, $p, q \in R$, and q installable in R. The pair $(p.n, v)$, where $v > p.v$,* challenges *q if q is not installable in any future F which is obtained by upgrading p to version v.*

Intuitively $(p.n, v)$ challenges q, when upgrading p to a new version v without touching any other package makes q not installable. This permits to pinpoint critical future upgrades that challenge many packages and that might therefore need special attention before being pushed to the repository. An example is given in Fig. 9.

```
Package: foo                                  Package: baz
Version: 1.0                                  Version: 1.0
Depends: bar (<= 3.0) | bar (>= 5.0)          Depends: foo (>= 1.0)

Package: bar
Version: 1.0
```

Fig. 9. Package `bar` challenges package `foo` for versions in the interval $]3.0, 5.0[$

The problem in deciding these properties is that any repository has an infinite number of possible futures. The two properties we are interested in belong to the class of so-called *straight* properties. For this class of properties it is in fact sufficient to look at a finite set of futures only which cover all of the problems that may occur in any future. One can show [5] that it is sufficient to look at futures where no package has been removed and new packages have been introduced only when their name was already mentioned in R, and where all new versions of packages have no conflicts and no dependencies. For any package there is an infinite space of all future version numbers, however, there is only a finite number of equivalence classes of these with respect to observational equivalence where the observations are the constraints on versions numbers used in R.

In reality, the definition of a future is more involved than the one given in Def. 11. In almost all distributions, packages are in fact not uploaded independently from each other but are updated together with all other packages stemming from the same *source package*. The complete definition of a future also takes into account a notion of *clusters* of packages, which are in our case formed by all binary packages stemming from the same source. Def. 13 has to be adapted accordingly, by allowing for all packages in the same cluster as p to be upgraded.

The full version of the algorithms in presence of package clusters, together with their proof of soundness, can be found in [5].

The top challenging upgrades in Debian Lenny found by our tool are listed in Table 5. Regularly updated reports on outdated Debian packages are available as part of the distribution quality assurance infrastructure.[8]

[8] `http://edos.debian.net/outdated.php`, retrieved January 2013

Table 5. Top 13 challenging upgrades in Debian lenny

Source	Version	Target Version	Breaks
python-defaults	2.5.2-3	≥ 3	1079
python-defaults	2.5.2-3	$2.6 \leq . < 3$	1075
e2fsprogs	1.41.3-1	any	139
ghc6	6.8.2dfsg1-1	$\geq$ 6.8.2+	136
libio-compress-base-perl	2.012-1	$\geq$ 2.012.	80
libcompress-raw-zlib-perl	2.012-1	$\geq$ 2.012.	80
libio-compress-zlib-perl	2.012-1	$\geq$ 2.012.	79
icedove	2.0.0.19-1	$>$ 2.1-0	78
iceweasel	3.0.6-1	$>$ 3.1	70
haskell-mtl	1.1.0.0-2	$\geq$ 1.1.0.0+	48
sip4-qt3	4.7.6-1	$>$ 4.8	47
ghc6	6.8.2dfsg1-1	6.8.2dfsg1+ $\leq . <$ 6.8.2+	36
haskell-parsec	2.1.0.0-2	$\geq$ 2.1.0.0+	29

With the same philosophy of identifying the impact of repository evolutions, it is important for quality assurance to be able to spot easily whether a new release has introduced new errors, and one particular error that affects FOSS distributions is the introduction of new incompatibilities among packages that were co-installable in a previous version. At first sight, identify such errors seems unfeasible: one would need to enumerate all possible sets of incompatible packages in the new distribution, and then check whether they were already incompatible in the previous release. Since there are 2^n package sets in a distribution with n packages, and n is in the tens of thousands, this approach is unfeasible. Recent work has shown that by introducing a notion of *cover* for incompatible package sets, it is actually possible to identify all such new errors in a very limited amount of time [21].

5 Related Work

Upgrade Simulation. Incompleteness and poor expressivity are just some of the issues that might be encountered while upgrading FOSS-based systems [18,24]. Several other issues can be encountered during actual package deployment, due to the unpredictability of *configuration scripts* execution on target machines. The formal treatment of those scripts is particularly challenging due to the fact that they are usually implemented in languages such as shell script and Perl, which are Turing-complete languages that also heavily rely on dynamic features such as shell expansions.

Model-driven techniques [12] have been applied to first capture the syntax and semantics of common high-level actions performed by configuration scripts in popular FOSS distributions, and then to instrument package deployment with simulators able to predict a significant range of upgrade failures before the actual target machine is affected [22,48,14,28].

Packages and Software Components. Packages share important features with *software components* [50,38], but exhibit also some important differences. On the one hand, packages, like components, are reusable software units which can be combined freely by a system administrator; they are also independent units that follow their own development time-line and versioning scheme.

On the other hand, packages, unlike what happens in many software component models, cannot be composed to build a larger component, and it is not possible to install more than one copy of a given package on a given system. Furthermore, installation of packages, and execution of software contained in packages, acts on shared resources that are provided by the operating system, like creating files on the file system, or interacting through the systems input/output devices. As a consequence, packages may be in conflict with each other, a phenomenon which is not (yet?) commonplace for software components.

Software components come with an interface describing their required and provided services. In the case of packages, requirements and provided features are given by symbolic names (either names of packages, or names of abstract features) whose semantics is defined separately from the package model. For instance, a policy document may describe how an executable must behave in order to provide a feature `mail-transport-agent`, or an external table will tell us which symbols have been provided in version 1.2.3 of library `libfoo`.

Packages and Software Product Lines. Issues similar to dependency solving are faced by semi-automatic configurators for software product lines [16]: they too have dependencies and conflicts, this time among features, and need to find a subset of them that work well together. Independently from packaging work, the application of SAT solving to feature selection has been investigated, among others, in [34].

The analogy between software product lines (SPL) and package repositories have been recently observed in other works, that explicitly show how to map one formalism into the other and vice-versa. The goal is to allow sharing of tools and techniques between the two worlds. The mapping from software product lines, based on the feature diagram formalism, to package repositories has been established in [26]; whereas a converse mapping, from Debian packages to SPL, has been more recently proposed in [32].

Packages and the Cloud. The idea of relying on automated tools to configure a software system based on (i) a repository of components and (ii) a user request to satisfy, can be applied in contexts larger than a single machine. The idea can in fact be extended to networks of heterogeneous machines, where each machine is associated to a specific package repository, and to higher-level services that span multiple machines and might induce inter-dependencies (and conflicts) among them.

This approach has been recently explored in the context of the Aeolus project [27], which directly tries to apply constraint solving to network and cloud settings and, with a slightly narrower but more easily automatable scope, also in the context of the Engage system [31].

References

1. Abate, P., Boender, J., Di Cosmo, R., Zacchiroli, S.: Strong dependencies between software components. In: ESEM 2009: 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 89–99 (2009)
2. Abate, P., Di Cosmo, R.: Predicting upgrade failures using dependency analysis. In: Abiteboul, S., Böhm, K., Koch, C., Tan, K.L. (eds.) Workshops Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, Hannover, Germany, April 11-16, pp. 145–150. IEEE (2011)
3. Abate, P., Di Cosmo, R., Treinen, R., Zacchiroli, S.: Mpm: a modular package manager. In: CBSE 2011: 14th International ACM SIGSOFT Symposium on Component Based Software Engineering, pp. 179–188. ACM (2011)
4. Abate, P., Di Cosmo, R., Treinen, R., Zacchiroli, S.: Dependency solving: a separate concern in component evolution management. Journal of Systems and Software 85(10), 2228–2240 (2012)
5. Abate, P., Di Cosmo, R., Treinen, R., Zacchiroli, S.: Learning from the future of component repositories. In: CBSE 2012: 15th International ACM SIGSOFT Symposium on Component Based Software Engineering, pp. 51–60. ACM (2012)
6. Abate, P., Di Cosmo, R., Treinen, R., Zacchiroli, S.: Learning from the future of component repositories. Science of Computer Programming (2012) (to appear)
7. Abate, P., Di Cosmo, R., Treinen, R., Zacchiroli, S.: A modular package manager architecture. Information and Software Technology 55(2), 459–474 (2013)
8. Albert, R., Jeong, H., Barabási, A.L.: Error and attack tolerance of complex networks. Nature 406(6794), 378–382 (2000)
9. Argelich, J., Le Berre, D., Lynce, I., Marques-Silva, J., Rapicault, P.: Solving Linux upgradeability problems using boolean optimization. In: LoCoCo: Logics for Component Configuration. EPTCS, vol. 29, pp. 11–22 (2010)
10. Artho, C.V., Di Cosmo, R., Suzaki, K., Zacchiroli, S.: Sources of inter-package conflicts in debian. In: LoCoCo 2011 International Workshop on Logics for Component Configuration (2011)
11. Artho, C.V., Suzaki, K., Di Cosmo, R., Treinen, R., Zacchiroli, S.: Why do software packages conflict? In: MSR 2012: 9th IEEE Working Conference on Mining Software Repositories, pp. 141–150. IEEE (2012)
12. Bézivin, J.: On the unification power of models. SOSYM 4(2), 171–188 (2005)
13. Boender, J.: Efficient computation of dominance in component systems (Short paper). In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 399–406. Springer, Heidelberg (2011)
14. Cicchetti, A., Di Ruscio, D., Pelliccione, P., Pierantonio, A., Zacchiroli, S.: A model driven approach to upgrade package-based software systems. In: Maciaszek, L.A., González-Pérez, C., Jablonski, S. (eds.) ENASE 2008/2009. CCIS, vol. 69, pp. 262–276. Springer, Heidelberg (2010)
15. Clayberg, E., Rubel, D.: Eclipse Plug-ins, 3rd edn. Addison-Wesley Professional (December 2008)
16. Clements, P., Northrop, L.: Software product lines. Addison-Wesley (2002)
17. Cosmo, R.D., Boender, J.: Using strong conflicts to detect quality issues in component-based complex systems. In: Padmanabhuni, S., Aggarwal, S.K., Bellur, U. (eds.) ISEC, pp. 163–172. ACM (2010)
18. Crameri, O., Knezevic, N., Kostic, D., Bianchini, R., Zwaenepoel, W.: Staged deployment in mirage, an integrated software upgrade testing and distribution system. SIGOPS Oper. Syst. Rev. 41(6), 221–236 (2007)

19. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM 7(3), 201–215 (1960)
20. Des Rivières, J., Wiegand, J.: Eclipse: a platform for integrating development tools. IBM Systems 43(2), 371–383 (2004)
21. Vouillon, J., Di Cosmo, R.: Broken sets in software repository evolution. In: ICSE 2013. ACM (to appear, 2013)
22. Di Cosmo, R., Di Ruscio, D., Pelliccione, P., Pierantonio, A., Zacchiroli, S.: Supporting software evolution in component-based FOSS systems. Science of Computer Programming 76(12), 1144–1160 (2011)
23. Di Cosmo, R., Lhomme, O., Michel, C.: Aligning component upgrades. In: Drescher, C., Lynce, I., Treinen, R. (eds.) LoCoCo 2011 International Workshop on Logics for Component Configuration, vol. 65, pp. 1–11 (2011)
24. Di Cosmo, R., Trezentos, P., Zacchiroli, S.: Package upgrades in FOSS distributions: Details and challenges. In: HotSWUp 2008: Hot Topics in Software Upgrades. ACM (2008)
25. Di Cosmo, R., Vouillon, J.: On software component co-installability. In: Gyimóthy, T., Zeller, A. (eds.) SIGSOFT FSE, pp. 256–266. ACM (2011)
26. Di Cosmo, R., Zacchiroli, S.: Feature diagrams as package dependencies. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 476–480. Springer, Heidelberg (2010)
27. Di Cosmo, R., Zacchiroli, S., Zavattaro, G.: Towards a formal component model for the cloud. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 156–171. Springer, Heidelberg (2012)
28. Di Ruscio, D., Pelliccione, P., Pierantonio, A., Zacchiroli, S.: Towards maintainer script modernization in FOSS distributions. In: IWOCE 2009: International Workshop on Open Component Ecosystem, pp. 11–20. ACM (2009)
29. Dolstra, E., Löh, A.: NixOS: A purely functional linux distribution. In: ICFP (2008) (to appear)
30. EDOS Project: Report on formal management of software dependencies. EDOS Project Deliverables D2.1 and D2.2 (March 2006)
31. Fischer, J., Majumdar, R., Esmaeilsabzali, S.: Engage: a deployment management system. In: Vitek, J., Lin, H., Tip, F. (eds.) PLDI, pp. 263–274. ACM (2012)
32. Galindo, J., Benavides, D., Segura, S.: Debian packages repositories as software product line models. towards automated analysis. In: Dhungana, D., Rabiser, R., Seyff, N., Botterweck, G. (eds.) ACoTA. CEUR Workshop Proceedings, vol. 688, pp. 29–34. CEUR-WS.org (2010)
33. Gonzalez-Barahona, J., Robles, G., Michlmayr, M., Amor, J., German, D.: Macro-level software evolution: a case study of a large software compilation. Empirical Software Engineering 14(3), 262–285 (2009)
34. Janota, M.: Do SAT solvers make good configurators? In: SPLC: Software Product Lines Conference, vol. 2, pp. 191–195 (2008)
35. Järvisalo, M., Berre, D.L., Roussel, O., Simon, L.: The international SAT solver competitions. AI Magazine 33(1) (2012)
36. Jenson, G., Dietrich, J., Guesgen, H.W.: An empirical study of the component dependency resolution search space. In: Grunske, L., Reussner, R., Plasil, F. (eds.) CBSE 2010. LNCS, vol. 6092, pp. 182–199. Springer, Heidelberg (2010)
37. LaBelle, N., Wallingford, E.: Inter-package dependency networks in open-source software. CoRR cs.SE/0411096 (2004)
38. Lau, K.K., Wang, Z.: Software component models. IEEE Trans. Software Eng. 33(10), 709–724 (2007)

39. Le Berre, D., Parrain, A.: On SAT technologies for dependency management and beyond. In: SPLC 2008: Software Product Lines Conference, vol. 2, pp. 197–200 (2008)
40. Le Berre, D., Rapicault, P.: Dependency management for the Eclipse ecosystem. In: IWOCE 2009: International Workshop on Open Component Ecosystems, pp. 21–30. ACM (2009)
41. Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. ACM Trans. Program. Lang. Syst. 1(1), 121–141 (1979)
42. Maillart, T., Sornette, D., Spaeth, S., von Krogh, G.: Empirical tests of zipf's law mechanism in open source linux distribution. Phys. Rev. Lett. 101, 218701 (2008), http://link.aps.org/doi/10.1103/PhysRevLett.101.218701
43. Mancinelli, F., Boender, J., Di Cosmo, R., Vouillon, J., Durak, B., Leroy, X., Treinen, R.: Managing the complexity of large free and open source package-based software distributions. In: ASE 2006: Automated Software Engineering, pp. 199–208. IEEE (2006)
44. Michel, C., Rueher, M.: Handling software upgradeability problems with MILP solvers. In: LoCoCo 2010: Logics for Component Configuration. EPTCS, vol. 29, pp. 1–10 (2010)
45. OSGi Alliance: OSGi Service Platform, Release 3. IOS Press, Inc. (2003)
46. Quinton, C., Rouvoy, R., Duchien, L.: Leveraging feature models to configure virtual appliances. In: Proceedings of the 2nd International Workshop on Cloud Computing Platforms, CloudCP 2012, pp. 2:1–2:6. ACM, New York (2012), http://doi.acm.org/10.1145/2168697.2168699
47. Raymond, E.S.: The cathedral and the bazaar. O'Reilly (2001)
48. Ruscio, D.D., Pelliccione, P., Pierantonio, A.: EVOSS: A tool for managing the evolution of free and open source software systems. In: Glinz, M., Murphy, G.C., Pezzè, M. (eds.) ICSE, pp. 1415–1418. IEEE (2012)
49. Steuer, R.E.: Multiple Criteria Optimization: Theory, Computation and Application. Wiley (1986)
50. Szyperski, C.: Component Software. Beyond Object-Oriented Programming. Addison-Wesley (1998)
51. Treinen, R., Zacchiroli, S.: Solving package dependencies: from EDOS to Mancoosi (2008)
52. Treinen, R., Zacchiroli, S.: Common upgradeability description format (CUDF) 2.0. Technical Report 3, The Mancoosi Project (November 2009), http://www.mancoosi.org/reports/tr3.pdf
53. Treinen, R., Zacchiroli, S.: Expressing advanced user preferences in component installation. In: IWOCE 2009: International Workshop on Open Component Ecosystems, pp. 31–40. ACM (2009)
54. Trezentos, P., Lynce, I., Oliveira, A.: Apt-pbo: Solving the software dependency problem using pseudo-boolean optimization. In: ASE 2010: Automated Software Engineering, pp. 427–436. ACM (2010)
55. Tucker, C., Shuffelton, D., Jhala, R., Lerner, S.: OPIUM: Optimal package install/uninstall manager. In: ICSE 2007: International Conference on Software Engineering, pp. 178–188. IEEE (2007)

Author Index