Universiteit Leiden
The Netherlands

Testing object Interactions

Grüner, A.

Citation
Grüner, A. (2010, December 15). *Testing object Interactions*. Retrieved from https://hdl.handle.net/1887/16243

Version:	Corrected Publisher's Version
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/16243

Note: To cite this publication please use the final published version (if applicable).

Testing Object Interactions

Proefschrift

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van de Rector Magnificus prof. mr. P.F. van der Heijden,
volgens besluit van het College vor Promoties
te verdedigen op woensdag 15 december 2010
klokke 13.45 uur

door

Andreas Grüner
geboren te Nettetal, Duitsland
in 1974

Promotiecommissie

Promoter:	Prof. dr. Frank S. de Boer
Co-promoter:	Dr. Marcello Bonsangue
	Dr. Martin Steffen *(Universitetet i Oslo)*
Referent:	Dr. Bernhard Aichernig *(Technische Universität Graz)*
Overige leden:	Prof. dr. Farhad Arbab
	Prof. dr. Joost N. Kok

Contents

1 Introduction **1**
1.1 Object-oriented programming languages . . . 2
1.1.1 *Java* . . . 3
1.1.2 C♯ . . . 4
1.2 Testing in the software development life-cycle . . . 4
1.3 Unit testing object-oriented software . . . 7
1.3.1 *JUnit* . . . 7
1.3.2 *jMock* . . . 10
1.3.3 *JMLUnit* . . . 14
1.4 Testing approach in this thesis . . . 16
1.5 Structure of the thesis . . . 17
1.6 Relation to my previous scientific work . . . 18

I Testing Sequential Components **19**

2 *Java*-like programming language – *Japl* **23**
2.1 Syntax . . . 24
2.2 Static semantics . . . 26
2.3 Operational semantics . . . 31
2.4 Extension by components: the *Japl* language . . . 37
2.4.1 Syntax . . . 38
2.4.2 Static Semantics . . . 40
2.4.3 Operational Semantics . . . 41
2.5 Traces and the notion of testing . . . 51

3 The test specification language **57**
3.1 Extension by expectations . . . 59
3.2 Syntax . . . 64
3.3 Static semantics . . . 66
3.4 Operational semantics . . . 73
3.5 Example . . . 76
3.6 Executability and input enabledness . . . 79

3.7 Satisfiability and completeness 81

4 Code generation **83**
4.1 Preprocessing 85
4.1.1 Labeling mechanism 85
4.1.2 Variable binding 93
4.2 *Japl* code generation 95
4.3 Generation of the test program. 102
4.4 Correctness of the code generation 103
4.5 Failure report and faulty specifications 111

5 Further possible extensions **115**
5.1 Specification classes 115
5.2 Programming classes 122
5.3 Subtyping and inheritance 126

II Testing Multi-threaded Components **135**

6 Concurrent programming language – *CoJapl* **139**
6.1 Syntax 139
6.2 Static semantics 140
6.3 Operational semantics 141

7 Specification language and code generation **151**
7.1 Syntax 154
7.2 Static semantics 157
7.3 Operational semantics 158
7.4 Test code generation 159

8 Concluding remarks **163**

Bibliography **167**

III Proofs **173**

Appendices **175**

A Subject reduction **177**

B Compositionality **181**

C Code generation **191**
C.1 Preprocessing 191
C.2 Anticipation 198
C.3 Correctness of the generated code 207

Summary **223**

Samenvatting **225**

Curriculum Vitæ **227**

List of Tables

2.1 Simple *Java*-like language: syntax 25
2.2 Simple *Java*-like language: type system (program parts up to stmts) 28
2.3 Simple *Java*-like language: type system (exprs) 30
2.4 Variable evaluation 33
2.5 Expression evaluation 33
2.6 Auxiliary notations 34
2.7 Simple *Java*-like language: operational semantics 36
2.8 *Japl* language : syntax 39
2.9 *Japl* language: type system (stmts) 40
2.10 Label check for incoming communication 45
2.11 Free variables 47
2.12 *Japl* language: operational semantics (ext.) 48
2.13 *Japl* language: traces 51

3.1 Specification language for *Japl*: syntax 65
3.2 Specification language for *Japl*: type system (stmts) 69
3.3 Specification language for *Japl*: operational semantics (external) . 74

4.1 Preprocessing: labeling and anticipation ($prep_{out}$) 89
4.2 Preprocessing: labeling and anticipation ($prep_{in}$) 90
4.3 Initial method and constructor code 99
4.4 Code-generation: method extension 99
4.5 Generation of *Japl* code ($code_{out}$) 101
4.6 Generation of *Japl* code ($code_{in}$) 102

5.1 Extension by specification classes: syntax 116
5.2 Extension by specification classes: type system (stmts) 118
5.3 Extension by specification classes: operational semantics 119
5.4 Extension by programming classes: syntax 123
5.5 Extension by programming classes: type system (stmts) 125
5.6 *Japl* with subclassing: syntax 127
5.7 *Japl* with subclassing: type system (stmts) 129
5.8 *Japl* with subclassing: operational semantics (int.) 131
5.9 Example: cross-border inheritance 132

5.10 *Japl* with subclassing: operational semantics (ext.) 133

6.1 *CoJapl* language: syntax 140
6.2 *CoJapl* language: type system (stmts) 142
6.3 *CoJapl* language: type system (exprs) 143
6.4 *CoJapl* language: operational semantics (internal, part 1) 145
6.5 *CoJapl* language: operational semantics (internal, part 2) 146
6.6 *CoJapl* language: operational semantics (external) 148
6.7 *CoJapl* language: traces 150

7.1 Specification language for *CoJapl*: syntax 155
7.2 *CoJapl* example specifications 156
7.3 Specification language for *CoJapl*: type system (stmts) 157
7.4 Specification language for *CoJapl*: operational semantics (external) 159
7.5 *CoJapl* code generation: mutual exclusion 162

C.1 Anticipation-valid code (static) 199
C.2 Anticipation-valid configurations (dynamic) 201
C.3 Simulation relation for statements 207
C.4 Well-typedness of dynamic specification code mc_{sl} 211

List of Figures

1.1 Software development process and testing levels 6
1.2 Novel testing approach 17

2.1 Notion of component 39

4.1 Testing framework 84

Chapter 1

Introduction

On Wednesday, the 30th September 2009, at 3.46am a data transmission problem, caused by a routine software update, resulted in a crash of an airline company's software system [58]. Specifically, the check-in systems of the airline and of some partner companies at more than 200 airports around the world were affected. The ground staff had to fall back on an older system of the 1970's – which basically consisted of writing passenger lists, boarding passes, and luggage tags, manually.

This system crash example joins a long list of more or less well-known software failures. Although the recent crash does not represent an overly spectacular software failure, yet it demonstrates that, on the one hand, developing complex software systems is still error-prone and, on the other hand, that the economy highly and increasingly depends on such software systems. Indeed, already in the late 1960's, software developers and scientists were aware of the discrepancy between the need for complex software systems and the difficulty of writing correct and reliable computer programs. It was F. L. Bauer who coined the term "*software crisis*" at the first NATO Software Engineering Conference in 1968 [52], in order to refer to the above mentioned software development dilemma. And in [23] Edsger W. Dijkstra stated in 1972:

> The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.

Since then various software development approaches, methodologies, and processes have been developed to come out of the crisis. Indeed, although the "silver bullet" has not been found yet, some progress has been made. The development of *high-level programming languages*, for instance, is based on the idea that language constructs abstracting from the details of the computer hardware allow a more efficient and less erroneous software development process. In particular, *object-*

oriented programming languages enjoy great popularity, as the object-oriented approach also facilitates the re-use of software components due to encapsulation and information hiding.

Moreover, to cope with the software dilemma, several standardized *software development process models* have emerged promising more predictable and more successful software projects. To achieve this, they subdivide the project into several smaller tasks and activities by the help of management and engineering techniques.

Besides efficiency, one key aspect of the software development processes in general is *quality assurance*. Although there has been much progress in recent years, proving the correctness of a software system by means of formal verification is still not feasible for most of the real world software products due to its high complexity. Thus, *testing* is still of prime importance in assuring the quality of software. In contrast to exhaustive methods for system verification and validation, testing aims at detecting faults, thus increasing confidence in the system under test [55, 71]. However, testing a complex product en bloc at the end of the development process, as it has been done in the early days of software testing, is not feasible anymore, either. In fact, to manage the complexity of modern software, testing has become a systematic operation, conducted on different levels and integrated into the software development process [33].

This thesis deals with testing at its lowest level, i.e., *unit testing*, of object-oriented software. In the remainder of this chapter, we first introduce the context of this work. Specifically, in the following sections, we will say some preliminary words regarding object-oriented languages, software development process models, and software testing. In particular, we will have a close look at some existing unit testing approaches for object-oriented languages. Finally, we will give a short overview of our testing approach and of the thesis.

1.1 Object-oriented programming languages

In object-oriented programming languages, a key concept is to *encapsulate* implementation details within so-called *objects* which represent an association of data with its operations. Specifically, the operations provide an *interface* between the object's data and its user rendering it unnecessary for the user to access the data directly. Due to this *information hiding*, the developer of the object, on the one hand, is free in choosing (or even in changing) the appropriate implementation for the data as well as the operations and, on the other hand, the user does not have to rely on a specific implementation but only on the interface. This idea can be traced back to the early 1960's. One of the first and probably most influential object-oriented languages was *Simula* [22], developed by Ole-Johan Dahl and Kristen Nygaard. *Simula* did not only introduce the concept of objects but also the notion of *classes*, used as "blue-prints" from which new objects can be instantiated. Moreover, the language supported *sub-classing* and *overriding*. That is, a class can inherit the data types and operations of another class and, beyond that,

it may re-define such an inherited operation by providing a new implementation.

Inspired by *Simula*, Alan Kay led the development of *Smalltalk* in the 1970's [30]. With *Smalltalk* Kay introduced the term *object-oriented programming* to express the pervasive use of objects and messages passing. Indeed, in *Smalltalk* everything is an object, including classes which can be created and modified dynamically.

As for the mainstream software application development, the object-oriented programming approach had its break-through in the early 1990's largely due to C^{++}, developed by Bjarne Stroustrup [67]. The programming language C^{++}, originally named "C with Classes", can be considered as an extension of the language C by object-orientation. Stroustrup developed C^{++} with the intention to make *Simula*'s object-oriented features available for real word software applications, since *Simula* was too slow for practical use. In fact, C regarded as a middle-level language, was and still is one of the most popular programming languages due to its execution speed.

The high performance of nowadays computer hardware, however, allows to use more high level computer languages also for most of the mainstream software applications. As a consequence, lots of high level programming language and scripting language with support for object-orientation have been developed. In the following, we will discuss two widely used representatives in more detail, namely *Java* and $C^{\sharp}$.

1.1.1 Java

Java is an object-oriented class-based general-purpose programming language which was developed at Sun Microsystems by a team headed by James Gosling. It was first released in 1996 [21]. Aiming at embedded systems, *Java*'s predecessor, *Oak*, was considered to be derived from C^{++}. Due to the lack of portability, however, the team decided to design a completely new language. Though, the syntax of *Java* is still inspired by C and C^{++}.

In contrast to the lower-level language C^{++}, *Java* does not allow pointer arithmetics. Specifically, a reference to an object is not represented by a pointer to a specific memory cell. Moreover, the language supports automatic garbage collection, i.e., the programmer needs not to allocate or de-allocate memory for objects, explicitly. *Java* is not fully object-oriented as it supports base types for integer or boolean values, for instance. However, for each base type there exists a corresponding class in *Java*, as well.

Java class definitions can be bundled to so-called *packages* which facilitate the re-use of class libraries [66]. In particular, the *Java* runtime environment comes with a huge class library including, among other things, thread classes allowing for a concurrent flow of control.

A *Java* program is compiled to *Java bytecode* which is executed by the *Java virtual machine* (JVM). *Java* bytecode is generally platform-independent. There exist JVM implementations for many computers and devices. For instance, today almost every cell phone is equipped with a JVM.

1.1.2 C♯

The programming language C♯ [25], first released in July 2000, can be considered as Microsoft's answer to *Java*. Developed by Anders Hejlsberg, the author of Turbo Pascal and chief designer of Delphi, C♯ is also an object-oriented class-based general-purpose programming language whose syntax likewise resembles that of C^{++}. Beyond that, it shares many other features with *Java*, like automatic garbage collection and the support for multi-threading. A C♯ program, too, is compiled to bytecode, called *Common Intermediate Language* (CIL), which is to be executed by the *Common Language Runtime* (CLR).

Apart from many similarities, C♯ provides some additional features which do not exist in *Java*. In contrast to *Java*, for instance, C♯ does support memory address pointers in order to increase execution speed in time critical applications. However, to prevent pointers from becoming a general security leakage, they may only be used within blocks which are to be marked as unsafe; unsafe blocks, in turn, need appropriate permissions to run. In this context, C♯ developers distinguish code which exclusively relies on automated garbage collection from code which includes user-allocated memory usage by the terms *managed* and *unmanaged code*.

Furthermore, C♯ introduces the concept of *delegates*. A delegate is a reference to an object's method which, in particular, can be passed around via method call parameters and return values. Consequently, a delegate may be invoked like a conventional method, although the caller need not to know the object of the method. However, the invoked delegate itself may access the object's fields and other methods.

Though, summarizing, there certainly exist some differences between *Java* and C♯, currently their similarities prevail. Aiming at object-oriented language more generally, in this thesis, we want to abstract from specific, distinguishing features but concentrate on the common characteristics of both languages. To this end, we will define and use a small object-oriented language intended to capture the object-oriented concepts that both languages have in common.

1.2 Testing in the software development life-cycle

It has been said, that several models regarding the software development process have been developed. Let us quickly discuss the basic idea of these models where we are specifically interested in the involved testing activities. Generally, the goal is to find repeatable and predictable processes that improve productivity and quality. In particular, to get a grip on the complexity of such a project, it is divided into smaller tasks. To this end, most models distinguish roughly the following phases:

- planning phase

  Usually, a software development project starts with a planning phase. The most important task within this phase is the *requirements analysis* where

the customer's needs and requests are gathered in a systematic way. This may lead to feasibility studies and first estimations regarding the effort, costs, and time needed.

- design phase

  Within the design phase, the overall *architecture* of the software system is to be determined. A hierarchy of subsystems and components is identified, such that the development processes can be divided into smaller manageable parts. The results includes a *specification* for each of the system's component capturing its requirements and its collaboration with other components.

- implementation and testing phase

  Based on the specification results of the design phase, the components are implemented. Furthermore, the specification of a component should be used as reference for a *component or unit test* where the component is tested in isolation. Following the hierarchy of the architecture, components are integrated resulting into larger components which in turn have to be tested by means of *integration test* activities. The idea of integration testing is to check whether the integrated components *interact* with each other as specified. The final integration test is called *system test* where the complete system is integrated.

- deployment and maintenance phase

  After completing and integration-testing the system, it is subject to an *acceptance test* with the customer. By this test, the customer checks whether the software meets the original requirements. Finally, if the acceptance test was successful, the software has to be integrated into the customer's production environment. However, in general this is still not the end of the software life cycle. For, often problems or improvement suggestions arise only during the daily operational use resulting into bug tracking or further software enhancement tasks.

A good example of a software development model demonstrating the relation between the actual development acitivities and the correpsonding testing activities is represented by the *V-model* (also: VEE model). The origins of the V-model can be traced back to the early 1980's [19]. Compared to its predecessor, the waterfall model, it has an emphasize on quality assurance aspects. Specifically, for each development phase it introduces a testing phase in which the results of the corresponding development phase are tested. As can be seen in Figure1.1, the V-model's course of action is often graphically represented in form of a V, hence, the model's name. The horizontal dashed lines indicate that test cases of a specific development phase should be formulated during the development phase itself, already.

As mentioned earlier, this thesis focuses on unit testing. For, a common statement is the later a software failure is observed during the software development

Figure 1.1: Software development process and testing levels

life cycle the more cost effect is the finding of the corresponding defect. For instance, B. Boehm and V. Basili state in [18] that finding and fixing a software problem after delivery is generally 100 times more expensive then finding and fixing it earlier. On the other hand, according to Jones, 85% of software failures are introduced during the design and the (low-level) implementation phase [39]. Therefore, low-level testing, i.e., unit testing, seems to have a key position for efficiency and quality in the software development process. This may be a reason why unit testing enjoys such a prominent role in agile software development processes like *extreme programming* [11]. For instance, concerning the extreme programming methodology, all code must have unit tests and all code must pass all unit tests before it can be released. In the following we will discuss three existing unit testing frameworks.

Before we discuss some exemplary unit testing approaches of the object-oriented world in the next section, however, let us first fix a terminology pertaining to testing that we will use in this thesis. Specifically, we will resort to corresponding definitions given in [61] (see also [63]).

Definition 1.2.1 (Errors, defects, and failures)**:** If a software developer makes an *error* (mistake), this results in a *defect* (fault, bug) in the code. If a defect in the code is executed, it may become observable in terms of a *failure*, i.e., the system may fail to do what it should do (or do something it should not do).

1.3 Unit testing object-oriented software

Unit testing in general represents the idea of automatically testing small program fragments, i.e., a unit or component, by executing a *test program* which incorporates the unit under test [62]. In the context of procedural or functional programming languages, a unit is often considered to be a procedure or, respectively, a function. Concerning object-oriented languages the smallest unit is used to mean an object or a class (depending on whether one wants to refer to dynamical, or respectively, static aspects). To test a unit, a test program *interacts* with the unit and possibly investigates the resulting change of the program state. While in a procedural or functional language these interactions are usually carried out in terms of function or a procedure calls, an object-oriented test program investigates the unit by means of *method calls*. A central aspect of object-oriented programming, however, is that objects heavily rely on the interaction with other objects. Thus, most often, an object-oriented unit test program has to ensure the existence of several *collaborator* objects that are required to cooperate as assumed by the unit in order to enable the unit to fulfill its tasks. Due to this fact, it is generally accepted that, regarding object-oriented systems, unit testing coincides with integration testing or, at least, that the dividing line between the two testing activities is blurred (cf. [6] and [16], for instance). As a consequence, writing unit tests in an object-oriented setting is usually considerably more complex. However, a couple of testing frameworks exist that aim at unit testing object-oriented components. We will have a closer look at three of these frameworks. The first two frameworks are widely used, specifically by the extreme programming community. Despite its usefulness, the third framework is not so commonly accepted. All three frameworks aim at the *Java* programming language. However, similar approaches do exist also for C$^\sharp$ and other related object-oriented programming languages.

To allow for comparison, each framework is illustrated by a simple example, realizing the test of a *voting system*. The voting system is a component that, when activated by an initiator, collects a vote from a group of external voter objects, compiles a report, and returns it to the initiator. It can be used, for example, to detect termination of a group of objects.[1] In our example, the voter system is implemented by means of a class *Census* defining a method *conductVoting* which realizes the above mentioned voting procedure. In particular, the method expects a list of *Voter* objects which, in turn, yield their vote in terms of a return value of a method *vote*. An exemplary implementation is sketched in Listing 1.1.

1.3.1 JUnit

JUnit [41] is a unit and regression testing framework written by Kent Beck and Erich Gamma. It has its origin in Kent Beck's *SUnit* [10], a unit testing framework for *Smalltalk*, and by now many adoptions to other languages exist. The collection of *JUnit* derivatives is often referred to as *xUnit*.

[1] We will, however, restrict our considerations on sequential programs until Part II of the thesis.

Listing 1.1: The voter system

```
class Voter {
  public Boolean vote() {
    ...
    return(value)
  }
}

class Census{
  public Boolean conductVoting(List voters) {
    Boolean result = true;
    for (Voter v : List) {
      result = result && v.vote();
    }
   return(result)
  }

}
```

The intention behind *JUnit* is to encourage software developers to write and execute tests themselves instead of shifting the responsibility on to some other software tester [12]. Software testing small units of code should become part of the code writing process. To integrate unit testing into the code writing process, Beck and Gamma suggest a development paradigm which is often called *test-driven development* (*TDD*). *TDD* represents a cornerstone of *Extreme Programming* [11] and other agile development approaches. In TDD developers write code incrementally by extending the unit only by one small feature at a time. More specifically, first small test cases for the new feature are written and afterwards the corresponding code is implemented. This is followed by exercising the unit tests. Only if all tests terminate successfully, the developer goes on to extend the unit with the next feature. It is worthwhile to say that the unit should not only pass the new tests but also all the other test cases of previously implemented features are executed, i.e., after extending the unit with a new feature, the developer does *regression testing*.

To avoid obstruction of the developers flow of work, *xUnit* tests are written in the same programming language as the *production code*, hence, *JUnit* tests are written in *Java*. A test (case) in *JUnit* is basically a *Java* program which incorporates and executes the code of the unit under test to decide on success or failure. To this end, the *JUnit* framework consists of a small and simple set of *Java* classes which expedites and unifies the recurring tasks when writing test code. The recent transition from version 3.8.1 to 4.0 entailed some major changes concerning the implementation and the usage of *JUnit*. As the former version is still widely-used, we will sketch both versions in the following.

The developer writes tests in terms of *Java* methods. Typically she or he

implements

- an (optional) setup method which initialises the data needed for the test case (in *JUnit* terms: the *test fixture*),
- the actual test case methods consisting of the interaction with the unit under test and the test evaluation,
- and an (optional) tear down method to free resources which possibly have been reserved by the setup method.

In version 3.8.1 and former versions of *JUnit*, the above mentioned methods have to be implemented in a subclass of *org.junit.Test.TestCase*. The setup and the tear down method are realized by overwriting the methods *setUp()* and *tearDown()*. The test case methods are methods with an arbitrary name[2] and without any parameters. *JUnit* allows for defining several test case methods within a single test case class, which therefore share the same *tearDown()* and *setUp()* methods, i.e. all test case methods of one test case class will be executed against the same test fixture. However, every instance of a test case class always executes only one test case method. Thus, for each instance the developer has to designate the desired test case method. This can be done either statically or dynamically. In the first case, the developer has to overwrite the *TestCase*'s method *runTest()* which is expected to call the designated test case method. In the latter case the name of the desired method is passed to the test case instance via a parameter of the constructor. *JUnit* uses reflection to find and execute the corresponding method.

With *JUnit* it is possible to execute a batch of test case methods to realize automated regression testing. To this end, test case instances can be grouped to test suites (which again can be grouped to other test suites, allowing for a tree structure of test cases). Again, *JUnit* supports a static and a dynamic way to add test cases to a test suite. Either the developer adds a test case by passing a corresponding test case *instance* to an instance of *JUnit*'s *TestSuite* or he passes a test case *class* to a test suite which then will, again, use reflection to create and add instances of that class for every test case method within the class. In that case, however, *JUnit* uses a naming convention to find all test cases at runtime by name, i.e., all test case methods must start with *test*. Finally, one has to implement a static method *suite()* which returns a test suite that contains all test cases to be executed.

A test case method typically calls a method of the unit under test and checks afterwards the return value or the resulting side effect of the method call. For this purpose, *TestCase* provides a set of assertion methods with a boolean parameter which is used to decide on success or failure. The method *assertTrue*, for instance, expects a boolean expression that has to evaluate to true, otherwise the test is considered as failed.

[2]However, to be able to use some test automation provided by *JUnit*, methods names have to obey certain naming conventions discussed later.

The biggest change that came with version 4.0 was the usage of *Java*'s annotations [36]. By using these annotations, developers need not to subclass *TestCase*, anymore. Instead, they mark a method as a test case, a tear-down, a setup, or a suite method by annotating them with a certain keyword. Moreover, additional keywords for new features have been introduced. For instance, apart from the tear-down and setup keywords which enables one to create and, resp., remove a fixture for every instance of a test case class, there exist new keywords which allow to create and remove parts of the fixture only once for all instances of a single class.

One criticism on *JUnit* 3.8 was the poor support for testing exceptions. If one wanted to ensure that a certain exception is raised in a certain situation, it was necessary to write a test case method which catches the corresponding exception if it has been raised. On the other hand, if the exception was not raised, one had to call the *fail* method of the *JUnit* framework manually to indicate that the test has failed. In *JUnit* 4 this is no longer necessary. Instead one can annotate a test method with the expectation of a certain exception.

Listing 1.2 and Listing 1.3 show the voter example for the test framework of *JUnit* 3.8.x and *JUnit* 4.x, respectively. The test fixture consists of three voters, instances of anonymous sub-classes of *Voter* to allow for different voting results, and one census object which is the actual unit under test. The sole test case method calls *conductVoting* of the census object and passes the above mentioned voters. After that it checks whether the result of the method call is as expected.

Essentially, the *JUnit* framework knows only one test pattern: call a method, wait until it returns and check the outcome. However, sometimes the developer wants to test not only the outcome at the end of a method call but also wants to ensure some features about the *interaction* in between the invocation and the return. For instance, in our example, we would like to ensure that *conductVoting* does not come to the right voting result only by chance, but that it indeed inquired the involved voting objects, i.e., we would like to test, whether *conductVoting* calls the method *vote* of the voter objects. This is not possible to test with *JUnit*[3].

There exist other unit testing frameworks in the style of *JUnit*. For instance, currently the strongest competitor of *JUnit* is most likely *TestNG* [68]. However, all these frameworks suffer from the lack of interaction test support.

1.3.2 jMock

The *Java* library *jMock* [38], developed by Nathaniel Pryce et alia, is also used for unit and regression testing of *Java* programs.

The *jMock* approach follows the idea that for testing object oriented systems it is more appropriate to test the interactions among objects rather than to test

[3]Actually, it certainly is possible to test this with *JUnit*, as a *JUnit* test is a normal *Java* program. Hence, every test that one can write in *Java* in general, can be embedded in the *JUnit* framework. But the test of interactions is not supported by *JUnit* directly, which means that one has to write additional code to realize this kind of tests

Listing 1.2: Voter example: *JUnit* 3.8.x

```
import org.junit.Test.TestCase;

class CensusTestCase extends TestCase {
  Vector voters;
  Census census;

  protected void SetUp() {
    voters = new Vector({
      new Voter { Boolean vote() { return true } },
      new Voter { Boolean vote() { return false } },
      new Voter { Boolean vote() { return true } }
    });
    census = new Census();
  }

  public void testVoting() {
    Boolean result = census.conductVoting(voters);

    Assert.assertTrue(result == false);
  }

}
```

the change of the program state caused by the interactions [28]. For, in object orientation it is often not possible to observe the state due to encapsulation. Moreover, even if an undesired state change has been observed then in many cases one has to identify the causing interaction, anyway.

For *interaction-based testing* [27], the developer has to identify the interaction partners of the unit under test and replace them by so-called *mock objects* [45] (regarding interaction-based testing, see also [57] and [8]) The task of these mock objects is to mimic the original environment objects of the unit and at the same time to verify assertions about the occurring interactions. Replacing the environment object by tester objects also makes sure that the unit is tested in isolation, i.e. the test is insulated from other possible failures caused outside of the component. Finally, this approach supports TDD, as even units can be tested whose final environment objects do not yet exist in the production code.

Usually *jMock* is applied on top of the *JUnit* testing framework which means that the developer writes *JUnit* test but utilizes the *jMock* library to formalize a behaviour-based testing with mock objects. Thus, one writes *JUnit* test case classes (if used with *JUnit* 3.8 or less) as described above with the following differences:

- Within the setup method, instead of setting up the test fixture by constructing the unit's environment by means of objects of the production code, one

Listing 1.3: Voter example: *JUnit* 4.x

```
import org.junit.Test.TestCase;

class CensusTestCase {
  Vector voters;
  Census census;

  @Before
  protected void createVotersAndCensus() {
    voters = new Vector({
      new Voter { Boolean vote() { return true } },
      new Voter { Boolean vote() { return false } },
      new Voter { Boolean vote() { return true } }
    });
    census = new Census();
  }

  @Test
  public void conductVotingAndCheckResult() {
    Boolean result = census.conductVoting(voters);

    Assert.assertTrue(result == false);
  }

}
```

creates mock objects correspondingly. However, as in common *JUnit* tests the object under test is certainly instantiated from a class of the production code.

- Within the test case method, one, firstly, formalizes the expected interactions between the component under test and the mock objects. Then, second, the unit's method to be tested is invoked and, finally, the expectations are verified.

The *jMock* library's basic idea is to support the creation of mock objects and the formalization of the expectations. However, the authors soon realized that in particular the design of the API for the formalization of the expected behavior has to be chosen carefully, as otherwise formalizations easily become tricky and error-prone. Thus, the design of the library is based on what the authors call an *embedded domain-specific language* [29] (EDSL). The idea is to provide developers a language for specifying interface behavior (hence, a domain-specific language) in terms of *Java* expressions. A key concept for this is *jMock*'s call chain syntax, where each method call to a *jMock* object yields another *jMock* object (or even the callee itself) such that another method invocation can be appended

without the need of prefixing the callee's name. A small example might clarify this. Suppose, one wants to formalize the expectation that the unit under test calls a method *buy* of object *mainframe* exactly once with parameter equal to the constant *QUANTITY*. Moreover, *buy* shall return the object *ticket*. Then one can write

```
mainframe.expects(once())
  .method("buy")
  .with(eq(QUANTITY))
  .will(returnValue(ticket));
```

Note, that the determination of the return value *ticket* does not represent an assertion on the behavior of the unit but stipulates the committed mock object's behavior towards the unit.

Listing 1.4 shows the voter example in terms of a behavior-based testing approach formalized with the help of the *jMock* library on top of the *JUnit* 4 framework. First, a *Mockery* object, *context*, is created which represents the entry point to the library. Moreover, an array of three mock objects of the *Voter* class is created. In the test case method *conductVotingAndCheckBehaviorAndResult* the expectations are formalized; the method *vote* of every voter (mock) object has to be called once. Additionally the return values are stipulated. After a call to the object under test *census* the result is checked.

With the help of the *jMock* library, the previous *JUnit* test example (Listing 1.3) has been improved in that now also the calls to the voter objects can be checked. However, although the authors put much effort into the design of the library, even this small example shows that there is still much "syntax noise", as Freeman and Pryce called it. In [29] they investigated the possibilities to create an EDSL for a more abstract description of the unit's behavior in context of a general purpose language like *Java* and concluded that "on the whole, it's too hard to extend conventional host languages, the syntax and the low-level operations get in the way".

Nevertheless, the mock object approach seems promising and by now several other implementations for *Java* exist. For instance, *EasyMock* [24] is also a well-known mock object library for *Java*. *EasyMock* tries to reduce the syntax noise by following the record-play idea. To formalize the expectations one calls first the mock objects methods as it is expected from the unit under test. After this "recording" step, the mode of the mock object is changed such that the mock object now expects (and realizes) the same interaction again. A drawback of this approach is that *EasyMock* by default only supports the generation of mock objects for interfaces. Moreover, this approach suffers from less expressiveness compared to the *jMock* approach.

Finally, although testing the observable interface behavior of a unit means also a higher, more abstract approach, by now no mock object implementations for *Java* support the use of the results for further analyses.

Listing 1.4: Voter example: *jMock*

```
import org.jmock.Expectations;
import org.jmock.Mockery;
import org.jmock.integration.junit4.JMock;
import org.jmock.integration.junit4.JUnit4Mockery;

@RunWith(JMock.class)
class CensusTestCase {
  Mockery context = new JUnit4Mockery();
  final Voter voters[] = {context.mock(Voter.class),
    context.mock(Voter.class), context.mock(Voter.class)}

  Census census;

  @Before
  protected void createVotersAndCensus() {
    census = new Census();
  }

  @Test
  public void conductVotingAndCheckBehaviorAndResult() {
    context.checking(new Expectations() {{
        one(voters[0]).vote(); will(returnValue(true));
        one(voters[1]).vote(); will(returnValue(false));
        one(voters[2]).vote(); will(returnValue(true));
    }};

    result = census.conductVoting(voters);
    Assert.assertTrue(result == false);
  }

}
```

1.3.3 JMLUnit

By using the *Java Modeling Language* (*JML*) [44], the unit testing tool *JMLUnit* [37] allows for specifying unit tests on a higher and more abstract level than *JUnit* or *jMock* do. In particular, developers need not to write the test code on their own but it is generated by *JMLUnit*.

JML is a specification language for *Java* programs which is used to formally specify the interface behavior of a *Java* module. It is based on the design-by-contract [46] (*DBC*) approach in the style of the *Eiffel* programming language [26]. *Eiffel* provides language constructs for defining contracts between method callers and method callees. These constracts consist of program code stating pre- and postconditions of methods and invariants of classes. Formalizing contracts in terms

of executable code, firstly lowers the burden on the programmer who does not need to learn an additional specification language, and secondly, it means that violations of the contracts can be detected at runtime. Since *Java* does not support *DBC* originally, *JML* specifications are stated in terms of special *Java* annotations embedded in the unit under test. The *JMLUnit* tool, in turn, extracts these annotations in order to generate code for *JUnit* test cases.

To allow for more formal specifications than it is possible in *Eiffel*, *JML* additionally builds on ideas from model-based specification languages like VDM-SL [40] (see also [7]) and the Larch family [32]. In particular, *JML* enables one to define abstract models of classes and objects by declaring model (and ghost) variables and methods. These variables and methods are not accessible by the actual unit code but can only be referred to within the *JML* annotations. Usage of these abstract models within the contract definitions leads to more abstract, more formal specifications.

Typically, a method's *JML* specification precedes the actual method declaration; class invariants precede the field declarations of a class. All *JML* specifications are *Java* comments which start with the at sign (@). Essentially, a method specification consists of a precondition and a postcondition. Preconditions are boolean predicates that must hold before the method is called; postconditions must hold after the execution of the method call. This means, the responsibility for establishing the precondition lies with the caller of the method – the responsibility for establishing the postcondition lies with the method itself. In *JML* preconditions and postconditions consist of the *JML* keyword *requires*, respectively, *ensures* followed by a boolean expression. Boolean expressions in *JML* are similar to normal *Java* boolean expressions. However, within a boolean expression one may only call methods that are declared as *pure* methods, i.e. methods that have no side effects. Moreover, *JML* provides additional construct which allow for more abstract specifications. For instance, one can quantify expressions by using *\forall* or *\exist*.

Listing 1.5 shows the voter unit code annotated with *JML* specifications. Line 5-6 formalize the required postcondition: method *conductVoting* may only yield `true` if, and only if, all voter objects likewise yield `true`. We have to assume, however, that *vote* is a pure method. Moreover, note that we cannot express the requirement that *conductVoting* must call the *vote* method of each voter object. Nevertheless, compared to *JUnit*, the example test specification is rather clear and concise. So the question may arise why *JUnit* is much more often in use than *JML*. One fact which might prevent *JML* from gaining more acceptance is its use of mathematical expressions. That is, although embedded into the *Java* code, the requirements are formalized in terms of mathematical formulas. Thus, despite the advantages of a more abstract specification, software developers seem to become reluctant, when likewise rather formal expressions come into play.

Listing 1.5: Voter example: *JMLUnit*

```
class Census {

  /*@ public behavior
    @ requires voters != null && voters.length() != null;
    @ ensures \result == true <==>
    @ (\forall int i; 0<= i && i < voters.length();
                    voters[i].vote() == true)
    @*/
  public void conductVoting( Voters[] voters) {
    result = true;

    for (int i=0; i++, i <= voters.length()) {
      result = result && voters[i].vote();
    }
  }
}
```

1.4 Testing approach in this thesis

This thesis proposes a novel approach for unit testing object-oriented components. The idea is to combine the benefits of the aforementioned existing testing frameworks. In particular, similar to the *JUnit* framework, the new approach should be accessible for software developers, it should allow for behavior-based testing like the *jMock* framework, and, finally, similar to the *JML* framework it shall allow for more abstract, hence, clear and concise, formalizations of the test cases where the underlying framework is based on a formal background.

However, we neither want to define an EDSL nor do we want to embed a formal language by means of annotations into the programming language. Instead, we embark on a *language extension* strategy. That is, we define a new *test specification language* by extending the original programming language of the production code with additional specification constructs. The intention of these tailor-made specification constructs it to provide the possibility for specifying a desired behavior of the unit under test in an abstract way. Moreover, in order to get an executable program that realizes the corresponding unit test, the test framework proposal comes with a *test code generation* algorithm that automatically generates programming language code from a specification of the test specification language. The testing approach is sketched in Figure 1.2.

The new specification constructs should not allow to specify aspects of the unit's behavior that has no impact on the unit's environment anyway. In other words, the constructs must aim at the *observable* behavior (cf. [49] and [53]) of the unit, only. To investigate the observable aspects of a unit's behavior in general, we provide a formally defined object-oriented programming language that is derived from *Java* and C$^\sharp$. Specifically, the language will serve as the formal bedrock

Figure 1.2: Novel testing approach

of our testing approach. For, apart from the features inspired from the above mentioned unit testing frameworks, we additionally want to support the unit testing of *concurrent* components which makes a formal context essential.

1.5 Structure of the thesis

The structure of this thesis is as follows. After this introductory chapter, the thesis consists of three parts. The first part deals with unit testing in context of a sequential object-oriented programming language. In particular, in Chapter 2 a formal definition for the *Java*-like object-oriented programming language *Japl* is developed. This is followed by the introduction of the test specification language for *Japl* in Chapter 3 and the code generation algorithm in Chapter 4. Finally, the first part concludes with the discussion about possible extensions of, both, the programming language and the test specification language.

The second part suggests a *concurrency* extension of the testing introduced in the first part. Specifically, Chapter 6 proposes an extension of the *Japl* programming language with *thread classes*. Correspondingly, Chapter 7 deals with an extension of the test specification language and, additionally, it sketches a suggestion on how to adapt the code generation algorithm of Chapter 4 in order to account for the concurrency extension. Finally Chapter 8 presents a conclusion of the thesis.

The third part of the thesis consists of the proofs. A central contribution of the thesis is the *correctness proof* of the code generation algorithm of Chapter 4. Although the *Java*-like language *Japl* covers only some basic aspects of typical object-oriented programming languages, still supported features like object-orientation and classes considerably increase the complexity of the proofs, already.

Thus, instead of embedding the proofs into the text they are presented separately in order to improve readability. In particular, it should be possible to understand and follow most of the ideas in this thesis without the need to understand all proofs in their details.

1.6 Relation to my previous scientific work

Many ideas of this thesis are based on or have been drawn from ideas related to my scientific work carried out and published during my Ph.D. studies.

In [2] a sequential class-based object calculus is introduced where programs consist of class definitions and a single thread definition. Considering components as sets of class definitions (and possibly a thread definition), the calculus serves as the mathematical vehicle for investigating the possible interaction *traces*, i.e., sequences of interactions that may take place between a component and its environment. The class-based setting makes instantiation a possible component-environment interaction which allows to create unconnected groups of objects, called *cliques*. Regarding a simple notion of *observability*, a notion of equivalence on these interaction traces is formalized which captures the uncertainty of observation caused by the fact that the observer may fall into separate cliques.

A similar class-based object calculus but additionally equipped with the support for *multi-threading* and re-entrant *monitors* has been proposed in [4]. The idea is to capture re-entrant monitor behavior, the basic synchronization and mutex-mechanism of, e.g., multi-threaded Java. A main result is that re-entrant monitors entail additional uncertainty of observation wrt. monitor operations at the interface which are captured by may- and must-approximations for potential, respectively, necessary lock ownership.

In [5] a class-based object calculus is introduced which allows dynamic thread instantiation by the support of *thread classes*. Similar to object instantiation, thread instantiation, occurring as a component-environment interaction, may lead to unconnected groups of objects which again increases the uncertainty of observation. The work formalizes a trace semantics with a notion of observable equivalence which accounts for the observational blur due to cliques.

In place of the thread-based concurrency model propagated by languages like *Java* and C$^\sharp$, the work in [3] deals with object-oriented languages that introduces concurrency by means of asynchronous message passing. A corresponding object calculus is introduced capturing, furthermore, *futures* and *promises* which act as proxies for, or reference to, the delayed result from some piece of code. This allows to compare the concurrency model based on asynchronous message passing with the thread-based approach on a solid mathematical basis.

Based on the idea that the trace of interface interactions between a component and its environment may serve as a specification for the desired behavior of a component under test, in [20] an automated test driver generation for *Java* components is proposed. In particular, a specification language for specifying the desired behavior of a *Java* component is introduced. Moreover, the paper sketches an algorithm which allows to generate a *Java* test driver from such a specification.

Part I

Testing Sequential Components

In this main part of this thesis we will propose a component testing approach for *Java* components. The contribution is threefold. We will define a *test specification language* which allows to specify the desired behavior of a component in terms of expected communication with its environment, i.e., in terms of its *interface behavior*. Moreover, we will present an algorithm for *automatically generating* a test program from a given specification such that the program tests for a component's conformance to the specified interface behavior. To this end, we will first present a *formally defined programming language* which captures a subset of the *Java* language. In particular, we will provide a formal semantics for components of this language. This enables us to investigate and characterize the possible observable interface behavior of a component.

The characterization will help us to find an appropriate design of the specification language, which will be a careful balance between two goals: we will use programming constructs in *Java*-like notation that help the programmer to specify the interaction without having to learn a completely new specification notation. On the other hand, *additional* expressions in the specification language will allow to specify the desired interface behavior in a concise, abstract way, hiding the intricacies of the required synchronization code at the lower-level programming language. Moreover, the formal language will be used to formalize the code generation algorithm and to proof its correctness.

Chapter 2

Java-like sequential programming language – *Japl*

This chapter introduces a *Java*-like programming language which we will use for further investigations. The intention is to provide a language that, on the one hand, captures a reasonable subset of features many modern object-oriented general-purpose programming languages like *Java* and C♯ have in common and that, on the other hand, comes with a formal semantics which allows to reason about the language.

Certainly there exist a couple of formal languages already aiming at *Java* or *Java*-like languages. For instance, in [1] Abadi and Cardelli suggested a core calculus for object-oriented languages. In addition they provide several extensions and modifications that deal with certain language features. Their typed imperative object calculus impς has been extended by Gordon and Hankin with concurrency [31] and a modification of the concurrent object calculus in turn was extended with classes in [5, 4], and in particular in [64]. The above mentioned calculi can be considered as an object-oriented counter-part to the family of λ-calculi. In a very concise way, they capture certain general features that almost all object-oriented languages have in common. Although these approaches represent a very good basis for investigating object-oriented languages in general, the (intended) generalization has its price. For, the provided abstract syntax is quite different from *Java* or C♯. Hence, it is sometimes not easy to find a *Java* program that corresponds to a given program of one of these object calculi and vice versa. Moreover, some language features are considered as special cases of other features. For example, in [1] there is no distinction between fields and methods which means that not only fields but also methods can be updated. Again, aiming at object-oriented languages in general, this represents an elegant unification. Since we restrict our approach to C♯-like and *Java*-like languages, however, this kind of

design decisions entails unnecessary complications.

However, there exist other approaches for capturing object-oriented languages which are closer to *Java*. Two prominent examples are *Featherweight Java* (*FJ*) [34] and *Middleweight Java* (*MJ*) [15]. The original *FJ* does not deal so much with *Java*'s operational aspects but rather captures *Java*'s type system. Thus, it is mainly used for investigating subtyping, inheritance, generics, and the like. *MJ*, in contrast, can be seen as an extension of *FJ* with respect to many of *Java*'s operational features most notably *MJ* introduces many imperative features.

The language that we propose here lies somewhere between *FJ* and *MJ*. In particular, our language is class-based, i.e., a program basically consists of a set of class definitions from which objects can be instantiated at runtime. Each object comprises a set of fields (also known as instance variables) and a set of methods. Objects are referenced by names which can be passed around giving rise to aliasing. The language is imperative, fields and variables allow for destructive updates. Furthermore, recursive method calls are possible.

To simplify matters, we do not consider, however, subtyping and inheritance. We will discuss these features and other possible extensions of the programming language in Chapter 5. We also omit more specific concepts like interface definitions, anonymous classes, generics, delegations, and reflection. Furthermore, in this part we focus on a sequential setting, that is, the language only allows for a single-threaded flow of control.

The rest of this chapter is structured as follows. In the first three sections, we will present the syntax, the type system, and, respectively, the operational semantics of closed programs of our language. A *closed program* is a self-contained entity in the sense that its possible behavior is completely determined by the given program code. In Section 2.4, in contrast, we will extend the language with the notion of *components* which will allow programs to contain references to classes that are not defined within the program code but are assumed to be provided by the program's environment. Finally we will conclude this chapter by presenting our testing approach in context of the new language and compare it with traditional unit testing.

2.1 Syntax

The grammar of the *Java*-like programming language is given in Table 2.1. A program consists of a list of global variables, a set of classes, and a main program (or main body). Note, that due to simplicity, our language slightly differs from *Java* already on the program level in two aspects: first, *Java* does not provide a designated construct for specifying global variables but rather requires them to be introduced by static fields. Second, in *Java* also the main program is not represented by a special construct on the program level but is given by a static method with a special name. However, to keep the language small and simple, we omit static fields and methods. Adding special constructs also allows for a clearer separation of concerns.

p	$::= \overline{T}\,\overline{x};\ \overline{cldef}\ \{stmt;\ \texttt{return}\}$	program
$cldef$	$::= \texttt{class}\ C\{\overline{T}\,\overline{f};\ con\ \overline{mdef}\}$	class definition
con	$::= C(\overline{T}\,\overline{x})\{\overline{T}\,\overline{x};\ stmt;\ \texttt{return}\}$	constructor
$mdef$	$::= T\ m(\overline{T}\,\overline{x})\{\overline{T}\,\overline{x};\ stmt;\ \texttt{return}\ e\}$	meth. definition
$stmt$	$::= x = e \mid x = e.m(e,\ldots,e) \mid x = \texttt{new}\ C(e,\ldots,e)$	statements
	$\mid f = e \mid \varepsilon \mid stmt; stmt \mid \{\overline{T}\,\overline{x};\ stmt\}$	
	$\mid \texttt{while}\ (e)\ \{stmt\} \mid \texttt{if}\ (e)\ \{stmt\}\ \texttt{else}\ \{stmt\}$	
e	$::= x \mid f \mid \texttt{null} \mid \texttt{this} \mid \texttt{op}(e,\ldots,e)$	expressions

Table 2.1: Simple *Java*-like language: syntax

The language is strongly typed, in the sense that the definitions of variables, formal parameters, and methods always include a type. The set of possible types is denoted by T. The details about the type system will be given in the next section. Here it suffices to say that T comprises class names and additionally some base types like Booleans and integer, if necessary.

We assume a number of meta-variables: $C, D, \ldots$ range over the set of class names *CNames*; $x, y, \ldots$ range over the set of variables *VNames*; f ranges over the set of field names *FNames*; and m ranges over the set of method names *MNames*. To keep the definitions compact, we use $\overline{C}$ (or, respectively, $\overline{f}$ or $\overline{cldef}, \ldots$), for the, possibly empty, sequence $C_1\ \ldots\ C_n$. Similarly we use $\overline{e}$ for the comma-separated sequence $e_1, \ldots, e_n$ and $\overline{T}\,\overline{x}$; to abbreviate the sequence $T_1\,x_1; \ldots; T_n\,x_n$. In slight abuse of the sequence notation, we sometimes also use it within function applications (or judgments) to denote the sequence or the set which results from applying the function on each element of the original sequence.

A class is given by its name, its field declarations, exactly one constructor, and a set of method definitions. Again, for simplicity we do not deal with subtyping or inheritance here. We assume, furthermore, that all fields are private, i.e., every object can directly access its own fields, only.[1]

Like in *Java*, both constructor and method definitions provide a list of formal parameters as well as a body which in turn may introduce new local variables and which ends with a return term. The return term of a method always includes a return value (possibly the undefined reference `null`) whereas a constructor's return term never does. Consequently, the method definition is not only equipped with a name but also with a return type. Note, that, as in *Java*, the name of the constructor is always the name of the class itself.

A statement is either an assignment, the empty statement (denoted by ε), a sequential composition, a block statement, a while-loop, or a conditional statement. Only expressions can be assigned to fields. Variables can additionally be updated by the result of a method or a constructor call.

An expression, finally, is either a variable, a field, the undefined reference `null`,

[1] Note that *Java*'s accessibility modifier `private` has a slightly different meaning.

the self reference `this` or a built-in operation. We do not deal with the details of the built-in operations but we only assume them to exist and to be side-effect free. Furthermore, we consider constants to be built-in operations with arity zero.

Remark 2.1.1 (Set notation)**:** Although we write classes, fields, and method definitions in a sequential way, their order has no meaning. Thus, we treat some constructs rather as sets. In particular, we will sometimes use set operations like $mdef \in \overline{mdef}$ or $\overline{fdef}_1 \cup \overline{fdef}_2$ or even $cldef \in p$.

We conclude this section with a small example program written in our language. The program is given in Listing 2.1. It consists of a global variable, two class definitions for binary trees, and a short main program. The first class `Data` is used to represent some data. The second class `BinTree` represents the binary tree structure. The main program first creates a data and a tree node object. The data is stored in a locally and the tree node instance in the globally defined variable. Afterwards another data and another tree node object are created, where the first tree node is passed to the constructor of the second tree node building the left branch of the new tree node.

2.2 Static semantics

Our programming language is statically typed. Thus, well-formedness of a program implies that we can associate a type with each of the program's variables and names such that the type assignments are consistent with regards to certain typing rules. We use type mappings, Γ and Δ, to denote these type assignments; for instance $\Gamma(x)$ either yields the type associated to variable x or is undefined. The mapping Γ provides the typing information of names which only have a local scope like variables and fields. It has a stack structure: appending a typed variable $x{:}T$ to an existing local mapping Γ, separated by a comma, creates a new mapping equal to Γ but extended by the new x which might shadow a possibly existing x in Γ.

In contrast, the mapping Δ contains the typing information of globally accessible constructs, namely of classes. Also for global mappings, we express its extension by appending typed names. However, the global mapping is not stack structured, since all names of classes are assumed to be different, hence, we don't have to deal with shadowing.

To express well-typedness, we introduce two kinds of typing judgments. Well-typedness of an expression e is denoted by a judgment of the following form:

$$\Gamma; \Delta \vdash e : T \ .$$

More precisely, the judgment states that, under the assumption of some type assignments given by $\Gamma; \Delta$, the expression e is well-typed and, additionally, that e itself is of type T. In this regard, the pair of type mappings $\Gamma; \Delta$ represents an assumption about the typed names provided by the environment of the expression e and we will refer to it as a *typing context*. The second kind of typing judgments

Listing 2.1: Simple example: Binary tree

```
BinTree s;

class Data {
  Data() { return }
}

class BinTree {
  BinTree lbranch;
  BinTree rbranch;
  Data value;

  BinTree(Data v, BinTree l, BinTree r) {
    value = v;
    lbranch = l;  rbranch = r;
    return
  }

  BinTree getLeft() { return lbranch; }

  BinTree getRight() { return rbranch; }

  Data getData() { return value; }

  BinTree setData(Data v) {
    value = v;
    return this;
  }
}

{
  { Data v;
    v = new Data();
    s = new BinTree(v, null, null);
    v = new Data();
    s = new BinTree(v, s, null)
  };
  return
}
```

is used for expressing well-typedness of entire programs and its syntactical constituents up to statements. Well-typedness of such a code fragment s is denoted by a judgment of the following form:

$$\Gamma; \Delta \vdash s : \mathsf{ok}\ .$$

Note that, in contrast to expressions, s does not provide a type.

Finally, we use the typing judgments to formalize the typing rules of our language. We provide the typing rules in the form of inference rules where each

$$[\text{T-Prog}]\ \frac{\Gamma' = \Gamma, \overline{x}{:}\overline{T} \qquad \Delta' = \Delta, \mathit{cltype}(\overline{\mathit{cldef}}) \qquad \Gamma'; \Delta' \vdash \overline{\mathit{cldef}} : \mathsf{ok} \qquad \Gamma'; \Delta' \vdash \mathit{stmt} : \mathsf{ok}}{\Gamma; \Delta \vdash \overline{T}\ \overline{x};\ \overline{\mathit{cldef}}\ \{\mathit{stmt};\ \texttt{return}\} : \mathsf{ok}}$$

$$[\text{T-Class}]\ \frac{\Gamma' = \Gamma, \overline{f}{:}\overline{T}, \texttt{this}{:}C \qquad \Gamma'; \Delta \vdash \mathit{con} : \mathsf{ok} \qquad \Gamma'; \Delta \vdash \overline{\mathit{mdef}} : \mathsf{ok}}{\Gamma; \Delta \vdash \texttt{class}\ C\{\overline{T}\ \overline{f};\ \mathit{con}\ \overline{\mathit{mdef}}\} : \mathsf{ok}}$$

$$[\text{T-Con}]\ \frac{\Gamma' = \Gamma, \overline{x}{:}\overline{T}, \overline{x'}{:}\overline{T'} \qquad \Gamma'; \Delta \vdash \mathit{stmt} : \mathsf{ok}}{\Gamma; \Delta \vdash C\ (\overline{T}\ \overline{x})\{\overline{T'}\ \overline{x'};\ \mathit{stmt};\ \texttt{return}\} : \mathsf{ok}}$$

$$[\text{T-MDef}]\ \frac{\Gamma' = \Gamma, \overline{x}{:}\overline{T}, \overline{x'}{:}\overline{T'} \qquad \Gamma'; \Delta \vdash \mathit{stmt} : \mathsf{ok} \qquad \Gamma'; \Delta \vdash e : T}{\Gamma; \Delta \vdash T\ m(\overline{T}\ \overline{x})\{\overline{T'}\ \overline{x'};\ \mathit{stmt};\ \texttt{return}\ e\} : \mathsf{ok}}$$

$$[\text{T-VUpd}]\ \frac{\Gamma; \Delta \vdash e : \Gamma(x)}{\Gamma; \Delta \vdash x = e : \mathsf{ok}} \qquad [\text{T-FUpd}]\ \frac{\Gamma; \Delta \vdash e : \Gamma(f)}{\Gamma; \Delta \vdash f = e : \mathsf{ok}}$$

$$[\text{T-Call}]\ \frac{\Gamma; \Delta \vdash e : C \quad \Gamma; \Delta \vdash \overline{e} : \Delta(C)(m).\mathit{dom} \quad \Gamma; \Delta \vdash x : \Delta(C)(m).\mathit{ran}}{\Gamma; \Delta \vdash x = e.m(\overline{e}) : \mathsf{ok}}$$

$$[\text{T-New}]\ \frac{\Gamma; \Delta \vdash x : C \quad \Gamma; \Delta \vdash \overline{e} : \Delta(C)(m).\mathit{dom}}{\Gamma; \Delta \vdash \texttt{new}\ x = C(\overline{e}) : \mathsf{ok}}$$

$$[\text{T-Seq}]\ \frac{\Gamma; \Delta \vdash \mathit{stmt}_1 : \mathsf{ok} \quad \Gamma; \Delta \vdash \mathit{stmt}_2 : \mathsf{ok}}{\Gamma; \Delta \vdash \mathit{stmt}_1;\ \mathit{stmt}_2 : \mathsf{ok}}$$

$$[\text{T-Block}]\ \frac{\Gamma, \overline{x}{:}\overline{T}; \Delta \vdash \mathit{stmt} : \mathsf{ok}}{\Gamma; \Delta \vdash \{\overline{T}\ \overline{x};\ \mathit{stmt}\} : \mathsf{ok}}$$

$$[\text{T-While}]\ \frac{\Gamma; \Delta \vdash e : \mathsf{bool} \qquad \Gamma; \Delta \vdash \mathit{stmt} : \mathsf{ok}}{\Gamma; \Delta \vdash \texttt{while}\ (e)\ \{\mathit{stmt}\} : \mathsf{ok}}$$

$$[\text{T-Cond}]\ \frac{\Gamma; \Delta \vdash e : \mathsf{bool} \qquad \Gamma; \Delta \vdash \mathit{stmt}_1 : \mathsf{ok} \qquad \Gamma; \Delta \vdash \mathit{stmt}_2 : \mathsf{ok}}{\Gamma; \Delta \vdash \texttt{if}\ (e)\ \{\mathit{stmt}_1\}\ \texttt{else}\ \{\mathit{stmt}_2\} : \mathsf{ok}}$$

Table 2.2: Simple *Java*-like language: type system (program parts up to stmts)

rule's conclusion consists of a specific judgment. If an instance of a typing rule is derivable then the corresponding code fragment in the conclusion is well-typed.

Before we introduce the typing rules, we make up for the missing definition of the types T.

Definition 2.2.1 (Types)**:** The set of types T of the programming language is given by means of the following grammar:

$$T ::= U \mid (\mathit{MNames} \cup \mathit{CNames}) \rightharpoonup (U \times \ldots \times U \to U)$$
$$U ::= C \mid \texttt{bool} \mid \mathbf{B}$$

Thus, the set of types comprises class names, class types, a Boolean type and, if necessary , some additional base types $\mathbf{B}$. Class names are used as types for objects whereas classes are typed with regards to their provided interface: a class type T is a partial function that maps each of the class' method and constructor name to a pair consisting of the parameter types and the return type. For methods m of the corresponding class, we will use $T(m).\mathit{ran}$ and $T(m).\mathit{dom}$ to denote the projection onto the first and, respectively, onto the second element of the pair, i.e., on the method's parameter types and, correspondingly, its return type. We use the same notation for the constructor name C of the class, where $T(C).\mathit{ran}$ always equals C.

Let us now discuss the typing rules in detail. Table 2.2 deals with the typing rules for programs and their syntactical constituents up to statements. Rule T-Prog stipulates that a program is well-typed if its class definitions and its main statement are well-typed. The set of premises representing the type checks of the class definitions is subsumed by using the sequence notation. The assumed typing contexts of both, the class definitions and the main statement, are enriched by the typed global variables and classes. Note, in order to carry out the class definitions' type-checks it is necessary to extend the type context by all class types already, as the method bodies might contain references to program classes. To this end, the type of a class is determined by the auxiliary function $\mathit{cltype}(\mathit{cldef})$ which extracts the type from the class definition's signature. It is defined as follows:

$$\mathit{cltype}(\ \texttt{class}\ C\{\ \overline{T}\ \overline{f};\ \mathit{con}\ \overline{\mathit{mdef}}\ \}\ ) \stackrel{\text{def}}{=} (C{:}f_C)\ \text{, with}$$
$$f_C : (\mathit{MNames} \cup \mathit{CNames}) \rightharpoonup (U \times \ldots \times U \to U);$$
$$n \mapsto \begin{cases} (\overline{T}, C) & \text{if } n = C \text{ and } C(\overline{T}\ \overline{x})\{\overline{T'}\ \overline{x'};\ \mathit{stmt};\ \texttt{return}\} \in \overline{\mathit{mdef}} \\ (\overline{T}, T) & \text{if } n = m \text{ and } T\ m(\overline{T}\ \overline{x})\{\overline{T'}\ \overline{x'};\ \mathit{stmt};\ \texttt{return}\ e\} \in \overline{\mathit{mdef}} \end{cases}$$

We assume that all method names of a class are distinct and that no method has the name of its class. This ensures that the function *cltype* is well-defined.

Rule T-Class deals with well-typedness of a class. A class is well-typed if its constructor and method definitions are well-typed. The type context of the constructor and method type-checks are enriched by the fields defined within the

$$[\text{T-VAR}]\ \frac{\Gamma(x) = T}{\Gamma; \Delta \vdash x : T} \qquad [\text{T-FIELD}]\ \frac{\Gamma(f) = T}{\Gamma; \Delta \vdash f : T}$$

$$[\text{T-NULL}]\ \Gamma; \Delta \vdash \texttt{null} : C \qquad [\text{T-THIS}]\ \frac{\Gamma(\texttt{this}) = C}{\Gamma; \Delta \vdash \texttt{this} : C}$$

$$[\text{T-OP}]\ \frac{\Gamma; \Delta \vdash \overline{e} : dom(\Delta(\texttt{op})) \quad ran(\Delta(\texttt{op})) = T}{\Gamma; \Delta \vdash \texttt{op}(\overline{e}) : T}$$

Table 2.3: Simple *Java*-like language: type system (exprs)

class as well as by the special name `this`, typed by the corresponding class, since `this` can be used for self references within the constructor and method bodies.

According to the rules T-CON and T-MDEF, a constructor as well as a method definition is well-typed if the body statement is well-typed assuming a type context that is extended by the local variables and the formal parameters. For methods additionally the type of the returned expression is checked.

Regarding all the variants of assignments, we have to check that the left-hand side and the right-hand side of the equals sign are of the same type. As for method and constructor calls we additionally have to check the types of the actual parameters. The parameter checks are expressed in slight abuse of the sequence notation: the elements of the parameter sequence and the elements of the method's parameter type tuple are matched concerning their order, such that each pair gives rise to a corresponding typing judgment. In particular, both sequences have to be of the same length. The typing rule for constructor calls (T-NEW) stipulates that objects are typed by the name of their class.

A sequence of statements is well-typed if each sub-statement is well-typed. Similarly, a block statement is well-typed if its body statement is well-typed where the local typing context is extended by the new variables declared by the block statement.

While statements and conditional statements are well-typed if their sub-statements are well-typed and if their conditional expressions are Boolean expressions.

Table 2.3 deals with the typing of expressions. Types of variables, fields and `this` can be directly looked up in the local type context Γ. The empty reference `null` is of any class name type. As for the built-in operations, we assume a typing to be already included in the global context Δ. Applications of these operations are only well-typed if the actual parameters conform to the domain type of the operation. If so, the application is of the range type of the operation.

Definition 2.2.2 (Well-typedness)**:** A program p is *well-typed* if there exist a type mapping Δ such that the judgment

$$; \Delta \vdash p : \mathsf{ok}$$

is derivable by means of the deduction rules given in Table 2.2 and 2.3. In particular,

the deduction starts with an empty local type mapping. Therefore, well-typedness of a program p regarding a certain type context Δ is denoted by

$$\Delta \vdash p : \mathsf{ok}\ .$$

2.3 Operational semantics

Operational semantics [59] is a way to express the meaning of a programming language: for each language construct, the effect of its execution on an abstract machine is formalized. The operational semantics of our language will be given in form of a small-step semantics. This kind of semantics is based on the idea that a program execution is considered as a sequence of indivisible steps that manifest themselves in form of changes in the program's configuration. The small-step semantics stipulates what kind of changes may happen in a certain situation. It is often represented by a transition relation which in turn is described by an inference system where the conclusion of each inference rule determines a (parameterized) transition between two configurations. The concept of using an inference system to describe the computation step, also called structural operational semantics, goes back to Plotkin [56]. Before we take a closer look at the operational semantics' transition rules let us first discuss the constituents of a program configuration. A program configuration $(h, \mathsf{v}, \mathsf{CS})$ is a triple consisting of the current state of the heap h, the global variables v, and the call stack CS. The details about the elements of a program configuration are given in the following definition.

Definition 2.3.1 (Configuration)**:** Let the set of all possible values be denoted by Val including null as the semantical representation for `null`. We use partial functions from field names to values to represent the state of an object. More specifically, an object consists of the value of its fields and a reference to its class. Thus, we define

$$\mathit{Obj} \stackrel{\text{def}}{=} \mathit{CNames} \times \mathsf{F} \quad \text{with } \mathsf{F} \stackrel{\text{def}}{=} \mathit{FNames} \rightharpoonup \mathit{Val}$$

as the set of all possible objects. For an object $o \in \mathit{Obj}$ we use $o.\mathit{class}$ and $o.\mathit{fields}$ to denote the projection onto the first or, respectively, the second element of the pair.

Let N be the set of *object names*. The heap is represented by a partial function from object names to objects. We use

$$\mathsf{H} \stackrel{\text{def}}{=} N \rightharpoonup \mathit{Obj}$$

to denote the set of heaps.

Let $\mathsf{V} \stackrel{\text{def}}{=} \mathit{VNames} \rightharpoonup \mathit{Val}$ be the set of *variable functions*, i.e., partial functions from variables to values. The state of a program's global variables is represented by an element v of V.

The *call stack* consists of a list of *activation records* each capturing the local variables as well as the code fragment of a method instance that still has to be executed. More precisely, an activation record's code fragment is of the form:

$$mc ::= \mathit{stmt}^x;\ mc \mid \texttt{return}\ [e],$$

where the square brackets denote optional terms. The non-terminal $stmt^x$ represents an extension of the non-terminal $stmt$ given in Table 2.1 in that it includes a new auxiliary statement BE which is needed for a proper processing of block statements. The details will be explained later. If a method is about to return the control back to the method's callee then the corresponding method fragment of the activation record consists of the return term only. Moreover, the very first activation record might represent the execution of the main body, which explains the square brackets around the return expression (as the main body does not return a value).

The state of the local variables of a method instance is given by a list of variable functions μ. Each variable function v of that list represents the state of the variables of a single block statement. Additionally, the local variable function list of a method instance always includes a variable function for the method's parameters, which is the empty function $\mathsf{v}_\perp$ if the method does not provide any parameter:

$$\mu ::= \mathsf{v} \cdot \mu \mid \epsilon.$$

For the time being, we can distinguish two kinds of activation records. The top-most activation record of a call stack represents the method instance which is currently *active*, i.e., which is in fact currently in execution. These activation records are always of the form:

$$\mathsf{AR}^a ::= (\mu, mc).$$

All other activation records within a call stack represent method or constructor instances which haven't finished their execution but which have called another method or constructor that hasn't returned yet. Thus, the calling method instance is *blocked* waiting for the return value of the called method or constructor. These activation records carry an auxiliary statement in front of the actual code fragment:

$$\mathsf{AR}^b ::= (\mu, \texttt{rcv}\ x; mc).$$

The receive statement is accompanied by the variable that is to be updated by the return value of the called method or constructor.

Finally, we can give a definition for the call stack:

$$\begin{array}{rcl} \mathsf{CS}^a & ::= & \mathsf{AR}^a \circ \mathsf{CS}^b \\ \mathsf{CS}^b & ::= & \mathsf{AR}^b \circ \mathsf{CS}^b \mid \epsilon \\ \mathsf{CS} & ::= & \mathsf{CS}^a \end{array}$$

The set $Conf$ is the set of all configurations, i.e.

$$Conf \stackrel{\text{def}}{=} \mathsf{H} \times \mathsf{V} \times \mathsf{CS}.$$

Before we discuss the transition rules of the operational semantics we introduce some auxiliary functions. The first group of auxiliary functions deals with evaluating and updating the variables of a program. The definitions are given in Table 2.4. The function $eval_m$ looks up the value of a variable accessible within

$$
\begin{aligned}
eval_m(\mathsf{v} \cdot \mu, x) &\stackrel{\text{def}}{=} \begin{cases} \mathsf{v}(x) & \text{if } x \in dom(\mathsf{v}) \\ eval_m(\mu, x) & \text{otherwise} \end{cases} \\
eval(\mathsf{v}, \mu, x) &\stackrel{\text{def}}{=} eval_m(\mu \cdot \mathsf{v}, x) \\
vupd_m(\mathsf{v} \cdot \mu, x \mapsto v) &\stackrel{\text{def}}{=} \begin{cases} \mathsf{v}[x \mapsto v] \cdot \mu & \text{if } x \in dom(\mathsf{v}) \\ \mathsf{v} \cdot vupd_m(\mu, x \mapsto v) & \text{otherwise} \end{cases} \\
vupd(\mathsf{v}, \mu, x \mapsto v) &\stackrel{\text{def}}{=} (\mathsf{v}', \mu') \\
&\quad \text{where } \mu' \cdot \mathsf{v}' = vupd_m(\mu \cdot \mathsf{v}, x \mapsto v)
\end{aligned}
$$

Table 2.4: Variable evaluation

a method instance. Following the structure of the nested scopes of the local variables, $eval_m$ recursively walks through the list of variable functions and returns the first defined value of the variable. The function *eval* evaluates a variable in the context of, both, a global variable function and the local variable function list. To this end, it appends the global variable function to the local variable function list, and passes the result as input parameter to $eval_m$. This way, the local variable context is extended by the global variables while respecting the possible shadowing effect by some local variables.

Similarly, we introduce two functions for updating the variable state. The first function $vupd_m$ takes a local variable function list as well as a variable-value pair and updates the first variable function within the list which defines a value for this variable. The second function *vupd* updates a variable of a program by, again, extending the local variable function list with the global variables and applying $vupd_m$. Then it takes the result and separates the global variables from the local variable list again, in order to return the updated variable pair.

The variable evaluation functions are used in the definition of the semantics

$$
\begin{aligned}
[\![x]\!]_h^{\mathsf{v},\mu} &\stackrel{\text{def}}{=} eval(\mathsf{v}, \mu, x) \\
[\![\texttt{this}]\!]_h^{\mathsf{v},\mu} &\stackrel{\text{def}}{=} eval(\mathsf{v}, \mu, \texttt{this}) \\
[\![f]\!]_h^{\mathsf{v},\mu} &\stackrel{\text{def}}{=} \mathsf{F}(f) \text{ with } (C, \mathsf{F}) = h(eval(\mathsf{v}, \mu, \texttt{this})) \\
[\![\texttt{null}]\!]_h^{\mathsf{v},\mu} &\stackrel{\text{def}}{=} \mathsf{null} \\
[\![\mathsf{op}(\overline{e})]\!]_h^{\mathsf{v},\mu} &\stackrel{\text{def}}{=} \mathsf{op}([\![\overline{e}]\!]_h^{\mathsf{v},\mu})
\end{aligned}
$$

Table 2.5: Expression evaluation

of expressions in Table 2.5. For each expression the semantics either is undefined or yields an element of *Val* depending on a given heap h and variable context represented by the global variable function v and a local variable function list μ. The evaluation of a variable and the self reference `this` is realized by just applying the aforementioned evaluation function. A field is evaluated by looking up the value of the self reference `this` which, in turn, is used to get the corresponding object; then the object's field function yields the desired value. The keyword `null` and the built-in operations are evaluated to their semantic representations. Note that the semantical representation null of the keyword `null` must not be an object name, i.e., we require $\mathsf{null} \notin N$.

The last group of auxiliary notations, given in Table 2.6, are simple functions

$$
\begin{aligned}
\mathit{classes}_p &\stackrel{\text{def}}{=} \{C_1, \ldots, C_k\} \\
&\quad \text{where} \\
&\quad p = \overline{T}\,\overline{x};\ \texttt{class}\ C_1\{\ldots\}\ \ldots\ \texttt{class}\ C_k\{\ldots\}\ \{\mathit{stmt};\ \texttt{return}\} \\
\mathit{fields}_p(C) &\stackrel{\text{def}}{=} \overline{T}\,\overline{f} \\
&\quad \text{where}\ \texttt{class}\ C\{\overline{T}\,\overline{f};\ \mathit{con}\ \overline{\mathit{mdef}}\} \in p \\
\mathit{cbody}_p(C) &\stackrel{\text{def}}{=} \mathit{stmt}; \\
\mathit{cparams}_p(C) &\stackrel{\text{def}}{=} \overline{T}\,\overline{x} \\
\mathit{cvars}_p(C) &\stackrel{\text{def}}{=} \overline{T'}\,\overline{x'} \\
&\quad \text{where}\ \texttt{class}\ C\{\overline{T}\ \overline{f};\ \mathit{con}\ \overline{\mathit{mdef}}\} \in p \\
&\quad \text{and}\ \mathit{con} = C(\overline{T}\,\overline{x})\{\overline{T'}\,\overline{x'};\ \mathit{stmt};\ \texttt{return}\} \\
\mathit{mbody}_p(C,m) &\stackrel{\text{def}}{=} \mathit{stmt};\ \texttt{return}\ e \\
\mathit{mparams}_p(C,m) &\stackrel{\text{def}}{=} \overline{T}\,\overline{x} \\
\mathit{mvars}_p(C,m) &\stackrel{\text{def}}{=} \overline{T'}\,\overline{x'} \\
&\quad \text{where}\ \texttt{class}\ C\{\overline{T}\ \overline{f};\ \mathit{con}\ \overline{\mathit{mdef}}\} \in p \\
&\quad \text{and}\ C'\ m(\overline{T}\,\overline{x})\{\overline{T'}\,\overline{x'};\ \mathit{stmt};\ \texttt{return}\ e\} \in \overline{\mathit{mdef}} \\
\mathit{Obj}^{C}_{p\bot} &\stackrel{\text{def}}{=} (C, \mathsf{F})\ \text{with}\ \mathit{dom}(\mathsf{F}) \stackrel{\text{def}}{=} \{f_1, \ldots, f_k\}\ \text{and} \\
&\qquad\qquad \mathsf{F}(f_i) \stackrel{\text{def}}{=} \mathit{ival}(T_i)\ \text{for all}, 1 \le i \le k \\
&\quad \text{where}\ T_1\ f_1;\ \ldots;\ T_k\ f_k = \mathit{fields}_p(C)
\end{aligned}
$$

Table 2.6: Auxiliary notations

that extract certain syntactical fragments of a given program. They are used in the definition of the operational semantics to keep the notation clear and concise. All functions have in common that they expect a well-formed program p as argument written as an index of each function. The function *classes* yields the set of class names defined in the program. Similarly, the functions *fields* yields the sequence of field names of a given class along with their types. The functions *cbody*, *cparams*, and *cvars* return the body of a class' constructor, its parameters, and its local variables, respectively. The functions *mbody*, *mparams*, and *mvars* do the same for methods. Note that the body of a method but not the body of a constructor includes the return term. The function $Obj^{C}_{p\perp}$ returns an object of class C where all the fields declared within C are set to the initial value of the corresponding type. We assume that for each base type T of our language there exists a certain constant, $ival(T)$, of type T which represents the initial value for variables of that type. In particular, we assume $ival(\mathsf{bool}) \stackrel{\text{def}}{=} \mathsf{false}$, $ival(\mathsf{int}) \stackrel{\text{def}}{=} 0$, and $ival(C) \stackrel{\text{def}}{=} \mathsf{null}$ for all class names C. Whenever we will use these auxiliary functions in the following, the considered program p will always be clear from the context, such that we will leave out the index notation.

The transition rules of the operational semantics are given in Table 2.7. Most of the rules share the same pattern. The leftmost statement of the topmost activation record is reduced which might cause a change of the heap and/or the values of the variables. Some of the rules can only be applied under certain conditions, represented by corresponding premises. The rule Ass deals with the assignment to a variable by merely updating the variable state. The rule FUpd updates a field. To this end, it looks up the object name currently stored in `this`. The name is used to get the corresponding object in the heap and to update its field function. The updated object in turn is used to update the heap. For functions f we use the notation $f[x \mapsto y]$ to denote a new function f' which is identical to f for all $z \in dom(f) \setminus \{x\}$ but which additionally maps x to y. Note, that this means either an extension of the original domain of f by the new element x or a modification of the image of x.

The rule Call extends the call stack by a new activation record consisting of the method body of the called method as well as of a new variable function. The variable function assigns the callee object name to `this`, the actual parameters to the formal parameters and it initializes the local variables. Moreover, the rule adds an auxiliary statement `rcv` x to the activation record of the calling method. After returning from the called method, this statement determines the variable which is to be updated by the return value. Note, that our language does not support the notion of exceptions. Thus, a method call whose callee expression evaluates to null gets stuck, as null is never part of the domain of the heap.

Processing a constructor call resembles very much the method call, but we have to add a new initial object to the heap and associate it to an object name which is not already in use. Moreover, we do not only copy the constructor body to the new activation record but we also add a return term with a return expression that yields the new object.

$$[\textsc{Ass}]\ \frac{(\mathsf{v}', \mu') = vupd(\mathsf{v}, \mu, x \mapsto [\![e]\!]_h^{\mathsf{v},\mu})}{(h, \mathsf{v}, (\mu, x = e; mc) \circ \mathsf{CS}^b) \rightsquigarrow (h, \mathsf{v}', (\mu', mc) \circ \mathsf{CS}^b)}$$

$$[\textsc{FUpd}]\ \frac{o = [\![\texttt{this}]\!]_h^{\mathsf{v},\mu} \quad (C, \mathsf{F}) = h(o) \quad h' = h[o \mapsto (C, \mathsf{F}[f \mapsto [\![e]\!]_h^{\mathsf{v},\mu}])]}{(h, \mathsf{v}, (\mu, f = e; mc) \circ \mathsf{CS}^b) \rightsquigarrow (h', \mathsf{v}, (\mu, mc) \circ \mathsf{CS}^b)}$$

$$[\textsc{Call}]\ \frac{\begin{array}{c} o = [\![e]\!]_h^{\mathsf{v},\mu} \quad C = h(o).class \quad \overline{T}\,\overline{x} = mparams(C, m) \quad \overline{T_l}\,\overline{x_l} = mvars(C, m) \\ \mathsf{v}_l = \{\texttt{this} \mapsto o, \overline{x} \mapsto [\![\overline{e}]\!]_h^{\mathsf{v},\mu}, \overline{x_l} \mapsto ival(\overline{T_l})\} \end{array}}{(h, \mathsf{v}, (\mu, x = e.m(\overline{e}); mc) \circ \mathsf{CS}^b) \rightsquigarrow (h, \mathsf{v}, (\mathsf{v}_l, mbody(C, m)) \circ (\mu, \texttt{rcv}\ x;\ mc) \circ \mathsf{CS}^b)}$$

$$[\textsc{New}]\ \frac{\begin{array}{c} o \in N \setminus dom(h) \quad h' = h[o \mapsto Obj_{\perp}^{C}] \quad \overline{T}\,\overline{x} = cparams(C) \quad \overline{T_l}\,\overline{x_l} = cvars(C) \\ \mathsf{v}_l = \{\texttt{this} \mapsto o, \overline{x} \mapsto [\![\overline{e}]\!]_h^{\mathsf{v},\mu}, \overline{x_l} \mapsto ival(\overline{T_l})\} \end{array}}{\begin{array}{c} (h, \mathsf{v}, (\mu, x = \texttt{new}\ C(\overline{e}); mc) \circ \mathsf{CS}^b) \rightsquigarrow \\ (h', \mathsf{v}, (\mathsf{v}_l, cbody(C);\ \texttt{return this}) \circ (\mu, \texttt{rcv}\ x;\ mc) \circ \mathsf{CS}^b) \end{array}}$$

$$[\textsc{BlkBeg}]\ \frac{\mathsf{v}_l = \{\overline{x} \mapsto ival(\overline{T})\}}{(h, \mathsf{v}, (\mu, \{\overline{T}\,\overline{x}; stmt\};\ mc) \circ \mathsf{CS}^b) \rightsquigarrow (h, \mathsf{v}, (\mathsf{v}_l \cdot \mu, stmt;\ \texttt{BE}\ mc) \circ \mathsf{CS}^b)}$$

$$[\textsc{BlkEnd}]\ (h, \mathsf{v}, (\mathsf{v}_l \cdot \mu, \texttt{BE}\ mc) \circ \mathsf{CS}^b) \rightsquigarrow (h, \mathsf{v}, (\mu, mc) \circ \mathsf{CS}^b)$$

$$[\textsc{Whl}_1]\ \frac{[\![e]\!]_h^{\mathsf{v},\mu}}{(h, \mathsf{v}, (\mu, \texttt{while}\ (e)\ \{stmt\}; mc) \circ \mathsf{CS}^b) \rightsquigarrow (h, \mathsf{v}, (\mu, stmt;\ \texttt{while}\ (e)\ \{stmt\}; mc) \circ \mathsf{CS}^b)}$$

$$[\textsc{Whl}_2]\ \frac{\neg[\![e]\!]_h^{\mathsf{v},\mu}}{(h, \mathsf{v}, (\mu, \texttt{while}\ (e)\ \{stmt\}; mc) \circ \mathsf{CS}^b) \rightsquigarrow (h, \mathsf{v}, (\mu, mc) \circ \mathsf{CS}^b)}$$

$$[\textsc{Cond}_1]\ \frac{[\![e]\!]_h^{\mathsf{v},\mu}}{(h, \mathsf{v}, (\mu, \texttt{if}\ (e)\ \{stmt_1\}\ \texttt{else}\ \{stmt_2\};\ mc) \circ \mathsf{CS}^b) \rightsquigarrow (h, \mathsf{v}, (\mu, stmt_1;\ mc) \circ \mathsf{CS}^b)}$$

$$[\textsc{Cond}_2]\ \frac{\neg[\![e]\!]_h^{\mathsf{v},\mu}}{(h, \mathsf{v}, (\mu, \texttt{if}\ (e)\ \{stmt_1\}\ \texttt{else}\ \{stmt_2\};\ mc) \circ \mathsf{CS}^b) \rightsquigarrow (h, \mathsf{v}, (\mu, stmt_2;\ mc) \circ \mathsf{CS}^b)}$$

$$[\textsc{Ret}]\ \frac{(\mathsf{v}', \mu_2') = vupd(\mathsf{v}, \mu_2, x \mapsto [\![e]\!]_h^{\mathsf{v},\mu_2})}{(h, \mathsf{v}, (\mu_1, \texttt{return}\ e) \circ (\mu_2, \texttt{rcv}\ x;\ mc) \circ \mathsf{CS}^b) \rightsquigarrow (h, \mathsf{v}', (\mu_2', mc) \circ \mathsf{CS}^b)}$$

Table 2.7: Simple *Java*-like language: operational semantics

The rules BlkBeg and BlkEnd deal with the introduction and removal of block variable functions due to a block statement. The rule BlkBeg does not only add a new variable function to the variable function list of the top most activation record but in the code of the record it also puts an auxiliary symbol (`BE`) at the end of the block statement, in order to mark the end of the block's scope. Then the counterpart of BlkBeg, namely BlkEnd, removes `BE` and its associated variables function when it is the topmost statement.

The while-loop is processed by either removing the while-loop from the active activation record or by extending it with a copy of the while-loop's body statement – depending on the evaluation of the while-loop's condition expression. In a similarly straightforward manner, the conditional statement is reduced to one of its sub-statements.

The RET rule is applied when the topmost activation record consists of a return term, only. The record as well as the receive statement of the calling activation record is removed such that the calling record becomes the topmost active record. Moreover, the caller's local variable list and the global variable list is updated by the return value.

Remark 2.3.2: Some rules of the operational semantics depend on the program code. Rule CALL, for instance, extends the call stack by the method body of the callee class. Thus, the transition rules are to be understood in context of a given program p and consequently the transition arrow should be annotated by the program: $\rightsquigarrow_p$. In most cases, however, we omit the annotation.

Definition 2.3.3 (Program execution)**:** Let

$$p \equiv \overline{T}\,\overline{x};\ \overline{cldef}\ \{stmt;\ \texttt{return}\}$$

be a syntactically correct and well-typed program of our language. A *program execution* of p is a finite sequence of reduction steps starting from the *initial configuration* of the program

$$c_{init}(p) \stackrel{\text{def}}{=} (h_\perp, \{\overline{x} \mapsto ival(\overline{T})\}, (\mathsf{v}_\perp, stmt;\ \texttt{return})),$$

where $h_\perp$ denotes the empty heap and $\mathsf{v}_\perp$ the empty local variable function. That is, both functions are completely undefined.

If we are interested neither in the exact length of a finite reduction sequence nor in its intermediate configurations we use a transition arrow annotated with the Kleene star,

$$c_{init}(p) \rightsquigarrow^* c,$$

expressing that there exists a finite sequence of reduction steps from the initial configuration to the configuration c or that both configurations are identical. In other words, we use the Kleene star annotation to refer to the reflexive and transitive closure of the semantics transition relation. If the call stack of c consists only of the last return statement of the main body, then we call c a *terminal configuration*. If otherwise c cannot be reduced any further, we call the configuration *faulty*.

2.4 Extension by components: the *Japl* language

As mentioned in the introduction, the basic idea for unit or component testing is to test the component in isolation. The production code that represents the *environment* of the component is replaced by some test code which investigates the component by interacting with it. Since we aim at a model-driven component testing approach where component tests are usually derived from formal, hence

rather abstract, specifications, we are in particular interested in testing techniques where a test does not rely on or aim at implementation details of the component but where a test only deals with the component's effects on its environment. In order to investigate the means by which a component might have an effect on its environment, we extend our *Java*-like language with constructs that allow for discriminating component and environment code. This is done by integrating a notion of *components* into our language. A component is basically a set of classes. Classes of one component can be imported by another component (or by the program). A crucial point is here that importing a class is not realized by importing the *code* of the class definition. Instead, we formalize the operational semantics of a program in absence of the code of imported classes. This enables us to identify the most general characterization of a component's influence on its environment, only assuming the interfaces of its classes.

In this section we will extend the *Java*-like language with the notion of components. The extended language will be used in subsequent chapters where we will refer to it as the *Japl programming language*. Conceptually, *Japl* supports components, in that it allows programs to import externally defined classes. This means, a program might instantiate and call methods of classes which are not part of the program's code but are assumed to exist in some other component. While the entailed syntax and typing modifications are quite simple, the operational semantics has to be given in form of an *open semantics*. In other words, we have to formulate the operational semantics without the code of the externally defined classes. The section is followed by a formal description of our testing approach given in context of our extended language.

2.4.1 Syntax

The extension of the syntax is very simple and straightforward, as can be seen in Table 2.8. We merely add a construct for declaring imported classes which introduces the name of the class only. For the sake of simplicity we do not introduce name spaces, but instead we assume that the names of all imported and all locally defined classes are different. The grammar introduces a new non-terminal symbol p' which replaces p of the former grammar. An element of p' is called a *component*. Thus, the program does not only import classes of other components but it constitutes a component itself. In particular, all components contain a main body. This is again due to simplicity, because otherwise we would have to differentiate components and programs (and in this case only component classes could be imported but a component could not import a program class). However, we will see in the operational semantics that only the main body of one single component is in execution.[2] In the following we will use the word *program* in order to refer to the component whose code is given and processed in the operational semantics. Note

[2]Assuming that each component provides its own main body is comparable to the widely used technique to equip a *Java* package with a static main method allowing for the stand-alone execution of the package due to testing or demonstration purposes.

Figure 2.1: Notion of component

that this doesn't necessarily mean that the main body of the program is executed, but maybe only the code of its classes is subject of the operational semantics. The program (component) might use classes of some *external* components. However, if the context is clear or if we don't want to be specific we sometimes speak only of components. The notion of components is depicted in Figure 2.1. It shows a program which defines classes C_1 to C_n as well as a main body and it additionally imports another class D from an unspecified external component. Thus, executing the given program, the operational semantics does not know the implementation but only the type of D.

p'	$::= \overline{impdecl};\ p$	program/component
$impdecl$	$::= \texttt{import}\ C$	import

Table 2.8: *Japl* language : syntax

$$[\text{T-PROG'}]\ \frac{\Gamma' = \Gamma, \overline{x}{:}\overline{T} \quad \Theta = \mathit{cltype}(\overline{\mathit{cldef}}) \qquad \Gamma;\Delta \vdash \overline{\mathit{impdecl}} : \mathsf{ok} \quad \Gamma';\Delta,\Theta \vdash \overline{\mathit{cldef}} : \mathsf{ok} \quad \Gamma';\Delta,\Theta \vdash \mathit{stmt} : \mathsf{ok}}{\Gamma;\Delta \vdash \overline{\mathit{impdecl}};\ \overline{T}\,\overline{x};\ \overline{\mathit{cldef}}\ \{\mathit{stmt};\ \mathtt{return}\} : \Theta}$$

$$[\text{T-IMPORT}]\ \frac{C \in \mathit{dom}(\Delta)}{\Gamma;\Delta \vdash \mathtt{import}\ C : \mathsf{ok}}$$

Table 2.9: *Japl* language: type system (stmts)

2.4.2 Static Semantics

We want to allow "cross-importing", i.e., components should be able to mutually import their classes. To this end, we have to reformulate the typing judgment on the program/component level such that it does not just state the program's well-typedness but it also explicitly mentions the program's classes committed to its environment in terms of a type mapping Θ. Moreover, we require that the assumed type context Δ of a program's type check already includes the types of the imported (*assumed*) classes. In other words, a program is now type-checked in an assumption-commitment context as it can be seen in typing rule T-PROG' in Table 2.9. This is closely related to the required and provided interfaces in *UML* compoment diagrams[65].

Finally, we have to add a new rule for the import construct. However, since the import construct only mentions the name of the class but no further typing information, we only have to check whether the imported class name is in the domain of Δ. All other rules of Table 2.2 and Table 2.3 remain the same.

As open programs are now typed in an assumption-commitment context, we have to reformulate the well-typedness definition for program.

Definition 2.4.1 (Well-typedness)**:** An open program p' is well-typed if there exist type mappings Θ and Δ such that the judgment

$$;\Delta \vdash p' : \Theta$$

is derivable. Similar to Definition 2.2.2, we demand an empty local type context for p. Therefore, well-typedness of a program p regarding the assumption-commitment context Δ, Θ is denoted by

$$\Delta \vdash p' : \Theta\ .$$

Remark 2.4.2 (Accessibility of global variables)**:** The global variables of a component are not "published" in the component's commitment context. The important consequence is that global variables of a component are not accessible by other components but they are always global with respect to the defining component, only.

2.4.3 Operational Semantics

The introduction of the import construct leads to an under-specification of the program: only the names and the types of the imported classes are given but not the code. The consequence is that the semantics definition of a component now consists of two parts. The first part, called *internal semantics*, deals with *internal* computations only, i.e., computations which are completely independent of the imported classes but solely determined by the program's code. This part is given in form of a transition semantics which is almost identical to the one already given in Table 2.7. We only add a premise in rule CALL and in rule NEW which ensures that the called method or constructor, respectively, indeed belongs to a class of the given program code. As for rule NEW, this additional check is very simple, since the class name itself is part of the `new` statement. As for the method call, we have to find out, if the callee object, named o, is an instance of a program class. We will see later, however, that the heap function stores information about objects of program classes, only. Therefore, the check can be easily realized by adding the premise $o \in dom(h)$.[3]

Labeled transition system. The second part of the operational semantics is called *external semantics*. It deals with computation steps that involve an (instance of an) external class. These steps must be handled differently as we do not have the code of the external classes: if, for example, the program calls a method of an external class, then we cannot use rule CALL, because lacking the method's code we cannot copy its body on the call stack in order to execute it afterwards. Instead, due to synchronous message passing, no further internal computations are possible right after the corresponding transition has been taken, as the transition gives away the control to an external component. We call this kind of transitions *outgoing* communication or computation steps. Right after an outgoing computation step, no transition can be taken unless it involves the return of the control back to the program. Generally speaking, we call transitions that entail the transition of control from an external component to the program *incoming* communication or *incoming* computation steps. In our method call example such an incoming communication could be either an immediate return from the called method of the external component or a call of a method of the program. Intuitively speaking, outgoing communication is due to a program statement, incoming communication is caused by an external component.

The external semantics is formalized by a *labeled transition system*, that is, a transition system where each transition is annotated with a label. Each transition of the external semantics represents an *interface communication*, i.e., a communication between the program and an external component and the transition's label holds the details about this communication. We have already seen that the

[3]In fact, it is not even necessary to add these checks, as $h(o).class$ in rule CALL and $cparams(C)$ in rule NEW are anyway undefined for an external object o, and, resp., class C. Adding the premises though makes the requirement more explicit.

different kinds of interface communications can be divided up into two groups. On the one hand, outgoing communications are provoked by the program giving the control to the external component. There exist three constructs in the programming language that may cause a hand-over of the control:

- a method call of an instance of an imported class,
- a constructor call of an imported class, and
- a return from a program method that was previously called by an external component.

To indicate the outgoing character, the transition labels of these communications are decorated with an exclamation mark (!). On the other hand, there are three possible kinds of communication that pass the control from an external component on to the program:

- a method call of an instance of a program class,
- a constructor call of a program class, and
- a return from a method of an imported class which was previously called by the program.

The labels of these incoming communication steps are decorated with a question mark (?). Summarizing, the possible interface interactions are method calls and returns as well as constructor call and returns which are either outgoing or incoming. Additionally, for each interface communication we explicitly have to point out the involved object names which pass the interface for the first time during the program execution. To this end, each communication label that propagates new names is equipped with a ν-binder indicating the new names introduced by the label. In particular, the ν-binder provides these names in terms of a type mapping Θ, since we are also interested in their types. The reason for this will be given later. Thus, the set of communication, or transition, labels a is given by the following grammar:

$$a ::= \gamma? \mid \gamma!$$
$$\gamma ::= \langle \mathit{call}\ o.m(v,\ldots,v)\rangle \mid \langle \mathit{new}\ C(v,\ldots,v)\rangle \mid \langle \mathit{return}\ (v)\rangle \mid \nu(\Theta).\gamma,$$
$$\text{where } o \in N, v \in \mathit{Vals}, \text{ and } \Theta \text{ is a type mapping.}$$

Restrictions due to realizability. The outgoing method call example above has shown, that the exact reaction of the external component is not determined; in general it is not known whether the component immediately returns a value or calls back a program method. Apart from the types, the program also has no control on the values that are involved in an incoming communication. In

fact, this is why we need a labeled transition system to formalize the external semantics: the details about an incoming communication cannot be deduced from the program code but are introduced by the communication label. More generally speaking, regarding incoming communications, the *operational semantics* is non-deterministic – although the *programming language Japl* in itself is deterministic. Despite the absence of the code of the external components, however we want to restrict the non-determinism of the operational semantics such that we exclude incoming communication which could not be carried out by any component that is written in our language *Japl*. This entails a number of requirements which have an influence on the structure of the transition rules:

well-typedness: Since *Japl* is strongly typed, a valid program written in *Japl* could never implement a wrongly typed method or constructor call. This fact implies a dual requirement for communication imposed by an external component. Specifically, the semantics shall only permit incoming calls of methods and constructors that are indeed provided by the program code. This requirement is often phrased as "no message-not-understood error". Also the number and the types of the incoming call's actual parameters must comply with the method or, respectively, constructor definition.

Likewise, the typing rules of our language ensure that the return value provided by a method body is always of the return type stipulated in the method's signature. Again, the dual requirement is that the semantics has to exclude wrongly-typed incoming return values.

consistent information flow: Components do not share variables but values can only be communicated between two components via method or constructor calls (and its corresponding returns). In particular, a component can only use an object if either it is an instance of one of the component's own classes or if the class-providing component has previously passed the corresponding object name in context of a method or constructor call between the two components. Thus, the external semantics has to ensure that an incoming communication may mention an object name of a program class only if previously the program has passed the name to the component by means of an outgoing communication.

consistent control flow: We have explained that an outgoing communication disables the reduction of the program unless an incoming communication occurs. Dually, the external semantics may only allow an incoming communication if the external component indeed has the control at that moment.

balance: The syntax definition of our language allows a return term only to appear exactly once at the very end of method and constructor bodies. That is, the execution of such return term is always preceded by a call of the corresponding method or constructor. For the external semantics, this means that an incoming return may only happen, if previously a matching outgoing call was invoked.

To meet the first two requirements, transitions of the external semantics are formalized in an assumption-commitment context, similar to the program's typing judgments of the static semantics. In the external semantics, however, the contexts also have a dynamic aspect, in the sense that the commitment context Θ does not only include the committed class names but also the names of their instances that have been passed on to the component during the execution of the program. Dually, the assumption context Δ consists of the component's class and object names passed on to the program. Thus, transitions representing computation steps of the external semantics will follow the following scheme:

$$\Delta \vdash c : \Theta \xrightarrow{a} \Delta' \vdash c' : \Theta',$$

where a is the label describing the communication and c is the configuration of the program prior to the transition. We used primed versions of Δ, Θ , and c in the transition scheme to indicate that the configuration as well as the context might change due to the transition. In particular, since the name contexts, Θ and Δ, keep track of the object names that passed the interface, each transition of the external semantics causes an extension of the context by exactly the names that are mentioned in the ν-binder of the transition's communication label. The transition rules of the external semantics are again formalized in terms of a deduction system where the conclusion of each rule consists of a transition scheme. Moreover, most of the rules are equipped with some premises defining the rule's application condition. In particular, we have to add certain premises to meet the realizability requirements regarding incoming communication.

To ensure a consistent information flow, the transition rules for incoming communication require that each object name mentioned in the call or return, respectively, is either included in the commitment context, in the assumption context or in the name context of the communication label's ν-binder. The first two cases indicate that the program and the component have communicated the object earlier already; the third case represents the situation where the object name shows up at the interface for the first time. Since all names in the contexts are also accompanied by their types, the contexts can be used the check for well-typedness as well. To keep the definition of the external semantics concise we encapsulate the type and information flow check in an auxiliary notation of the following form:

$$\Delta \vdash a : \Theta,$$

which expresses that, within a context represented by Δ and Θ, the computation step represented by a conforms to well-typedness and a consistent information flow. The definition is given in Table 2.10. The rule T-CallI deals with the type and information flow check of a label that represent an incoming method call. This is basically carried out by the premises of the first row: the first premise states that the callee object is indeed an object of the program; the second and the third premise ensure the existence of the method within the callee class as well as the well-typedness of the actual parameters. Since we use the name contexts for the type check we ensure at the same time that all object names of the communication

$$[\text{T-CALLI}]\ \frac{C = \Theta(o) \quad \overline{T} = \Theta(C)(m).dom \quad \Theta, \Delta, \Delta' \vdash \overline{v}{:}\overline{T} \quad dom(\Delta') \perp dom(\Delta, \Theta) \quad dom(\Delta') \subseteq \overline{v}}{\Delta \vdash \nu(\Delta').\langle call\ o.m(\overline{v})\rangle ? : \Theta}$$

$$[\text{T-NEWI}]\ \frac{\Theta(C)(C).dom = \overline{T} \quad \Theta, \Delta, \Delta' \vdash \overline{v}{:}\overline{T} \quad dom(\Delta') \perp dom(\Delta, \Theta) \quad dom(\Delta') \subseteq \overline{v}}{\Delta \vdash \nu(\Delta').\langle new\ C(\overline{v})\rangle ? : \Theta}$$

$$[\text{T-RETI}]\ \frac{dom(\Delta') \perp dom(\Delta, \Theta) \quad dom(\Delta') \subseteq v}{\Delta \vdash \nu(\Delta').\langle return(v)\rangle ? : \Theta}$$

Table 2.10: Label check for incoming communication

label are mentioned in one of the name contexts, hence, a consistent information flow is assured. The premises of the second row check for well-formedness of the ν-bound name context Δ': first, the names which are claimed to pass the interface for the first time, must not be included in Δ or Θ. We use $\perp$ to express that two sets are disjunct. Second, the name context Δ' must not include more names than actually communicated by this interface communication.

The rule T-NEWI deals with an incoming constructor call and follows the scheme of rule T-CALLI. However, we need not to look up the name of the invoked class. Finally, the check of an incoming return label even only consists of the well-formdness check of the ν-bound name context.

Configurations. We have learned that extending the language with the notion of components leads to a new kind of method (and constructor) calls: besides internal calls, where caller and callee belong to the same component, we now also have external (or cross-border) calls, where caller and callee sit in different components. With respect to the configurations of a program, this means, we also have to introduce a second receive statement enabling us to distinguish activation records that are blocked due to an internal call from activation records that are blocked due to an external call. More precisely, regarding internal calls we keep using the receive statement that we have introduced previously. As for the external calls we use a similar receive statement but which is additionally annotated with the return type of the expected return value. Thus, we now have three types of activation records:

$$\mathsf{AR}^a ::= (\mu, mc) \qquad \mathsf{AR}^{ib} ::= (\mu, \texttt{rcv}\ x;\ mc) \qquad \mathsf{AR}^{eb} ::= (\mu, \texttt{rcv}\ x{:}T;\ mc).$$

For the sake of convenience we will also use AR for activation records in general and AR^b for internally or externally blocked activation record. Moreover, we will use AR^i for AR^a and AR^{ib} records, i.e., for activation records whose next upcoming reduction will be due to an internal step.

We redefine the call stack structure of configurations of components:

$$\mathsf{CS}^a ::= \mathsf{AR}^a \circ \mathsf{CS}^b \qquad \mathsf{CS}^b ::= \mathsf{CS}^{ib} \mid \mathsf{CS}^{eb}$$
$$\mathsf{CS}^{eb} ::= \mathsf{AR}^{eb} \circ \mathsf{CS}^b \mid \epsilon \qquad \mathsf{CS}^{ib} ::= \mathsf{AR}^{ib} \circ \mathsf{CS}^b$$
$$\mathsf{CS} ::= \mathsf{CS}^a \mid \mathsf{CS}^{eb}$$

Contrary to the call stack of a closed program, a call stack of an open program does not necessarily have an *active* activation record on top but the call stack can also be externally blocked due to an outgoing call. Note that externally blocked call stacks also include the *empty* call stack. The reason is that in some cases not the main body of the program but the main body of an external component is executed. Then, for instance at the very beginning of the execution, we start with a configuration that entails an empty call stack.

Furthermore, we introduce the notion of well-typedness of configurations. Similar to the typing judgments of open programs, well-typedness of a configuration is expressed by writing the configuration in an assumption-commitment context consisting of type mappings Δ and Θ. Before we present the definition for well-typed configurations, we define the free variables of activation record code.

Definition 2.4.3 (Free variables)**:** Assume code mc of an activation record. The expression $\mathit{fvars}(mc)$ denotes the set of *free variables* within mc. The function fvars is given for activation record code by the recursive definition shown in Table 2.11.

Definition 2.4.4 (Well-typed configuration)**:** Let $(h, \mathsf{v}, \mathsf{CS}) \in \mathit{Conf}$ be a configuration and Δ, Θ typing contexts. We say the configuration $(h, \mathsf{v}, \mathsf{CS})$ is well-typed in context of the assumed type mapping Δ and the committed type mapping Θ if the following properties hold:

1. For all $(C, f) \in \mathit{ran}(h)$, it is $\Theta \vdash C[\![\ldots]\!]$.
2. For all $(\mu, mc) \in \mathsf{CS}$ and for all $x \in \mathit{fvars}(mc)$, it is $x \in \mathit{dom}(\mathsf{v}) \cup \mathit{dom}(\mu)$.
3. For all $\mu_m \in \mathsf{CS}$, it is $[\![\texttt{this}]\!]_{_}^{\mu_m,_} \in \mathit{dom}(h)$.

Thus, well-typedness of configurations ensures that the class names of objects in the heap are indeed committed as names of classes. The second statement ensures that each variable mentioned in the code of an activation record is either a global variable or a local variable of the record. We use $\mathit{dom}(\mathsf{v}_1 \cdot \ldots \cdot \mathsf{v}_k)$ as abbreviation for $\mathit{dom}(\mathsf{v}_1) \cup \ldots \cup \mathit{dom}(\mathsf{v}_k)$. Finally, we are assured that the local variables of each activation record representing a method instance provide a value for the self-reference `this` which in turn refers to an object in the heap.

$$
\begin{array}{rcl}
fvars(\texttt{rcv}\ x; mc) & \stackrel{\text{def}}{=} & \{x\} \cup fvars(mc) \\
fvars(\texttt{rcv}\ x{:}T; mc) & \stackrel{\text{def}}{=} & \{x\} \cup fvars(mc) \\
fvars(x = e) & \stackrel{\text{def}}{=} & \{x\} \cup fvars(e) \\
fvars(f = e) & \stackrel{\text{def}}{=} & fvars(e) \\
fvars(x = e.m(e_1, \ldots, e_k)) & \stackrel{\text{def}}{=} & \{x\} \cup fvars(e) \cup_{i=1,\ldots,k} fvars(e_i) \\
fvars(x = \texttt{new}\ C(e_1, \ldots, e_k)) & \stackrel{\text{def}}{=} & \{x\} \cup_{i=1,\ldots,k} fvars(e_i) \\
fvars(stmt_1;\ stmt_2) & \stackrel{\text{def}}{=} & fvars(stmt_1) \cup fvars(stmt_2) \\
fvars(\{\overline{T}\ \overline{x};\ stmt\}) & \stackrel{\text{def}}{=} & fvars(stmt) \setminus \overline{x} \\
fvars(\texttt{while}(e)\{stmt\}) & \stackrel{\text{def}}{=} & fvars(e) \cup fvars(stmt) \\
fvars(\texttt{if}(e)\{stmt_1\}\ \texttt{else}\ \{stmt_2\}) & \stackrel{\text{def}}{=} & fvars(e) \cup fvars(stmt_1) \cup fvars(stmt_2) \\
fvars(\texttt{BE}) & = & \emptyset \\
fvars(\texttt{return}\ e) & = & fvars(e) \\
fvars(\texttt{return}) & = & \emptyset \\
fvars(x) & \stackrel{\text{def}}{=} & \{x\} \\
fvars(\texttt{this}) & \stackrel{\text{def}}{=} & \emptyset \\
fvars(f) & \stackrel{\text{def}}{=} & \emptyset \\
fvars(\texttt{null}) & \stackrel{\text{def}}{=} & \emptyset \\
fvars(\texttt{op}(e_1, \ldots, e_k)) & \stackrel{\text{def}}{=} & fvars(e_1) \cup \ldots \cup fvars(e_k)
\end{array}
$$

Table 2.11: Free variables

Transition rules. After this somewhat longer preliminary explanation, let us now have a closer look at the rules of the external semantics. They are given in Table 2.12. To improve readability, we distinguish conditions that express a real limitation of the rule's application from conditions that only introduce variables used to keep the definition short. The first kind of conditions are written as premises, the latter are written as side conditions. An exception is the introduction of the communication label a which is always listed as the first premise in every rule.

The first rule, CallO, implements an outgoing call. This rule must be applied only if the callee of the method is indeed an object of an external component. This is checked by assuring that the callee object name is an element of the domain of the name context Δ. In this sense, rule CallO is the counterpart of rule Call which deals with internal method calls, only. However, we do not put a method body on the call stack but instead we add a `rcv` statement to the current activation

$$[\textsc{CallO}]\ \frac{a = \nu(\Theta').\langle call\ o.m(\overline{v})\rangle! \quad o \in dom(\Delta)}{\begin{array}{c}\Delta \vdash (h, \mathsf{v}, (\mu, x = e.m(\overline{e}); mc) \circ \mathsf{CS}^b) : \Theta \xrightarrow{a} \\ \Delta \vdash (h, \mathsf{v}, (\mu, \mathtt{rcv}\ x{:}T;\ mc) \circ \mathsf{CS}^b) : \Theta, \Theta'\end{array}}$$

where $o = [\![e]\!]_h^{\mathsf{v},\mu}$, $\overline{v} = [\![\overline{e}]\!]_h^{\mathsf{v},\mu}$, $T = \Delta^2(o)(m).ran$, and $\Theta' = \mathsf{new}(h, \overline{v}, \Theta)$

$$[\textsc{NewO}]\ \frac{a = \nu(\Theta').\langle new\ C(\overline{v})\rangle! \quad C \in dom(\Delta)}{\begin{array}{c}\Delta \vdash (h, \mathsf{v}, (\mu, x = \mathtt{new}\ C(\overline{e}); mc) \circ \mathsf{CS}^b) : \Theta \xrightarrow{a} \\ \Delta \vdash (h, \mathsf{v}, (\mu, \mathtt{rcv}\ x{:}C; mc) \circ \mathsf{CS}^b) : \Theta, \Theta'\end{array}}$$

where $\overline{v} = [\![\overline{e}]\!]_h^{\mathsf{v},\mu}$ and $\Theta' = \mathsf{new}(h, \overline{v}, \Theta)$

$$[\textsc{RetO}]\ \frac{a = \nu(\Theta').\langle return(v)\rangle!}{\begin{array}{c}\Delta \vdash (h, \mathsf{v}, (\mu, \mathtt{return}\ e) \circ \mathsf{CS}^{eb}) : \Theta \xrightarrow{a} \\ \Delta \vdash (h, \mathsf{v}, \mathsf{CS}^{eb}) : \Theta, \Theta'\end{array}}$$

where $v = [\![e]\!]_h^{\mathsf{v},\mu}$ and $\Theta' = \mathsf{new}(h, v, \Theta)$

$$[\textsc{CalleI}]\ \frac{a = \nu(\Delta').\langle call\ o.m(\overline{v})\rangle? \quad \Delta \vdash a : \Theta}{\begin{array}{c}\Delta \vdash (h, \mathsf{v}, \mathsf{CS}^{eb}) : \Theta \xrightarrow{a} \\ \Delta, \Delta' \vdash (h, \mathsf{v}, (\mathsf{v}_l, mbody(C, m)) \circ \mathsf{CS}^{eb}) : \Theta\end{array}}$$

where $C = \Theta(o)$, $\overline{T}\ \overline{x} = mparams(C, m)$, $\overline{T'}\ \overline{x'} = mvars(C, m)$, and $\mathsf{v}_l = \{\mathtt{this} \mapsto o, \overline{x} \mapsto \overline{v}, \overline{x'} \mapsto ival(\overline{T'})\}$

$$[\textsc{NewI}]\ \frac{a = \nu(\Delta').\langle new\ C(\overline{v})\rangle? \quad \Delta \vdash a : \Theta}{\begin{array}{c}\Delta \vdash (h, \mathsf{v}, \mathsf{CS}^{eb}) : \Theta \xrightarrow{a} \\ \Delta, \Delta' \vdash (h', \mathsf{v}, (\mathsf{v}_l, cbody(C)) \circ \mathsf{CS}^{eb}) : \Theta\end{array}}$$

where $o \in N \setminus dom(h)$, $h' = h[o \mapsto Obj_{\perp}^{C}]$, $\overline{T}\ \overline{x} = cparams(C)$, $\overline{T'}\ \overline{x'} = cvars(C)$, and $\mathsf{v}_l = \{\mathtt{this} \mapsto o, \overline{x} \mapsto \overline{v}, \overline{x'} \mapsto ival(\overline{T'})\}$

$$[\textsc{RetI}]\ \frac{a = \nu(\Delta').\langle return(v)\rangle? \quad \Delta \vdash a : \Theta \quad \Delta, \Delta', \Theta \vdash v : T}{\begin{array}{c}\Delta \vdash (h, \mathsf{v}, (\mu, \mathtt{rcv}\ x{:}T;\ mc) \circ \mathsf{CS}^b) : \Theta \xrightarrow{a} \\ \Delta, \Delta' \vdash (h, \mathsf{v}', (\mu', mc) \circ \mathsf{CS}^b) : \Theta\end{array}}$$

where $(\mathsf{v}', \mu') = vupd(\mathsf{v}, \mu, x \mapsto v)$

Table 2.12: *Japl* language: operational semantics (ext.)

record which is annotated with the return type of the called method. Thus, the activation record becomes an externally blocked activation record. To determine the return type, we consult the name context Δ. Since, however, the object's type is a class name, we have to apply Δ twice to get the corresponding class type which in turn yields the return type of the method. We use Δ^2 to express the double application of Δ. The return type will be needed in rule RetI to check that the incoming return value matches the outgoing call.

Apart from modifying the call stack we also have to update the *committed* name context, since the outgoing call might involve the propagation of names of some program objects which haven't passed the interface previously. In order to find out these new object names we use an auxiliary function $\mathsf{new}(h, \overline{o}, \Theta)$. For a given set of objects $\overline{o}$ it consults the heap h in order to create a subset of $\overline{o}$ which consists only of instances of program classes accompanied by their types. Finally the current committed name context Θ is subtracted leaving only the new object names. Its definition is as follows:

Definition 2.4.5 (New names propagation)**:**

$$\mathsf{new}(h, \overline{o}, \Theta) \stackrel{\text{def}}{=} \{(o{:}C) \mid o \in \overline{o} \wedge o \in dom(h) \wedge C = h(o).class\} \setminus \Theta.$$

The rule NewO deals with the instantiation of a class of an external component. It resembles rule CallO but it is a bit simpler, as there is no need to look up the return type. Rule RetO processes a return statement. More specifically, it must only deal with outgoing returns, i.e., it must only be applied if the corresponding method has been called by an external component. This, however, is the case, if the next activation record on the call stack is an externally blocked record but, in particular, not an internally blocked record that waits for an internal return. Thus, the rule expects that the topmost activation record is followed by an externally blocked call stack.

The next three rules deal with incoming communications, namely with incoming method calls (CallI), incoming object creation (NewI), and incoming returns (RetI). All three rules implement the type and information flow consistency check as it has been previously explained. Moreover, all three rules assert consistency of the control flow, in that they start from a configuration with an externally blocked call stack. In Rule RetI, however, we do not use CS^{eb} in the judgment to express this requirement but we explicitly demand the annotated `rcv` statements on top of the call stack, as we use its annotated return type to check well-typedness of the return value in the communication label. Second, an annotated `rcv` statement on top also ensures the balance condition, as it indicates a preceded outgoing call (cf. rules CallO and NewO). After all, CS^{eb} also comprises the empty statement, hence, in contrast to Rule RetI, the Rules CallI and NewI can also be applied when the call stack is empty. This allows to execute the program through the use of an external main body.

Note that our approach for defining the external semantics of *Japl* is closely related to Tretmans' input-output transition systems[69]. This kind of labeled transition systems also distinguishes input and output labels in order to determine the direction of communication occurring between a component and its environment. Moreover, input-output transition systems are input-enabled which reflects the fact that an exact behavior of the environment is not specified. However, although *Japl* components are generally input-enabled as well we have made clear that realizability demands some restrictions regarding the possible interface communication. For instance, as already mentioned the balance condition forbids transitions with incoming return labels in certain situations.

Remark 2.4.6 (Renaming)**:** The introduction of new objects of an external class by means of an incoming communication is actually carried out by passing the object's *name* to the program. Although the label check for incoming communication ensures that the name of such an object indeed hasn't shown up *at the interface* previously, there might exist an *internal* object of the same name within the program already. In this case, we assume an implicit ad hoc renaming of the internal object. More precisely, we rename the object in the heap and at the same time change all corresponding references within

fields or variables of the component. As an important consequence in the following we will consider configuration equality always up to renaming of objects, only.

Note, that we do not have to arrange for capture-free renaming as object names are globally[4] unique.

An important feature of the operational semantics is that it preserves the well-typedness of configurations. This is formalized in the following lemma.

Lemma 2.4.7 (Subject reduction)**:** Assume $\Delta_0 \vdash p' : \Theta_0$ and $\Delta \vdash c : \Theta$ for a configuration c such that Δ and Θ represent extensions of Δ_0 and Θ_0, respectively.

1. If $c \rightsquigarrow_p c'$ then also $\Delta \vdash c' : \Theta$.

2. If $\Delta \vdash c : \Theta \xrightarrow{a}_p \Delta' \vdash c' : \Theta'$ then $\Delta' \vdash c' : \Theta'$. Moreover, Δ' and Θ' again represent extensions of Δ_0 and Θ_O.

Thus, the subject reduction lemma justifies the use of typing judgments for transition states of the external operational semantics.

Definition 2.4.8 (Program execution; program traces)**:** Let

$$p' \equiv \overline{impdecl}\ \overline{T}\ \overline{x};\ \overline{cldef}\ \{stmt;\ \texttt{return}\}$$

be an open program with $\Delta \vdash p' : \Theta$. We broaden the application of c_{init}, defined in Definition 2.3.3, such that we also apply it to open programs p'. However, in the context of open programs we call $c_{init}(p')$ an *active* initial configuration. Dually, we introduce a *passive* initial configuration

$$\overline{c_{init}}(p') \stackrel{\text{def}}{=} (h_\perp, \overline{x} \mapsto ival(\overline{T}), \epsilon),$$

where the call stack is not initialized with the main body of p' but it is empty, meaning that the main body of an external component is to be executed.

The execution of an open program is represented by a finite, possibly empty, sequence of internal and external transitions starting from one of its initial configurations. The sequence of communication labels arising from a program execution is called an *(observable interaction) trace* of the program. We use an annotated arrow $\stackrel{t}{\Longrightarrow}$ to represent a program execution that implements the trace t. The corresponding rules are given in Table 2.13. Thus, the execution of an open program represents the reflexive transitive closure of the internal and external transitions.

From now on, we deal with open programs only. Specifically, a closed program is considered to be an open program which does not import external classes. Therefore, the syntactical discrimination between open and closed programs by using different non-terminal symbols p and p' is not necessary anymore, instead we will use p also for open programs.

[4]With "global" we always mean global with respect to a certain component only, but not to the whole system consisting of all components that might be involved in a program execution.

$$[\textsc{Intern}]\frac{c \longrightarrow^* c'}{\Delta \vdash c : \Theta \overset{\epsilon}{\Longrightarrow} \Delta \vdash c' : \Theta}$$

$$[\textsc{Single}]\frac{\Delta \vdash c : \Theta \overset{a}{\rightarrow} \Delta' \vdash c' : \Theta'}{\Delta \vdash c : \Theta \overset{a}{\Longrightarrow} \Delta' \vdash c' : \Theta'}$$

$$[\textsc{Seqnc}]\frac{\Delta \vdash c : \Theta \overset{s}{\Longrightarrow} \Delta' \vdash c' : \Theta' \qquad \Delta' \vdash c' : \Theta' \overset{t}{\Longrightarrow} \Delta'' \vdash c'' : \Theta''}{\Delta \vdash c : \Theta \overset{st}{\Longrightarrow} \Delta'' \vdash c'' : \Theta''}$$

Table 2.13: *Japl* language: traces

2.5 Traces and the notion of testing

We have seen that the execution of an open program gives rise to a sequence of its interface interactions that occur during the execution. We use these sequences, called traces, to define a semantics of a program which characterizes the program's possible effect on its environment, i.e. on other, external components. Based on this *trace semantics*, we will formalize a notion of testing that builds the formal basis of our testing approach.

Definition 2.5.1 (Trace Semantics)**:** We introduce three semantic functions

$$[\![\cdot]\!]^a_{trace}, [\![\cdot]\!]^p_{trace}, \text{ and } [\![\cdot]\!] : \Delta \vdash p : \Theta \rightharpoonup \mathscr{P}(a^*),$$

such that for well-typed open programs p with $\Delta \vdash p : \Theta$ we define

$$\begin{aligned}
[\![\Delta \vdash p : \Theta]\!]^a_{trace} &\overset{\text{def}}{=} \{s \in a^* | \Delta \vdash c_{init}(p) : \Theta \overset{s}{\Longrightarrow} \Delta' \vdash c' : \Theta'\}, \\
[\![\Delta \vdash p : \Theta]\!]^p_{trace} &\overset{\text{def}}{=} \{s \in a^* | \Delta \vdash \overline{c_{init}}(p) : \Theta \overset{s}{\Longrightarrow} \Delta' \vdash c' : \Theta'\}, \text{ and} \\
[\![\Delta \vdash p : \Theta]\!] &\overset{\text{def}}{=} [\![\Delta \vdash p : \Theta]\!]^a_{trace} \cup [\![\Delta \vdash p : \Theta]\!]^p_{trace}
\end{aligned}$$

Thus, the first two semantics definitions yield the set of traces that can be implemented by a program p with some interacting external components, where the program is either considered as a passive or as an active component, respectively. The third definition then consists of all traces that can be realized by the program with any external component.

We want to use the notion of traces to formalize what it means to test a component on the basis of its interface behavior. However, for a better understanding of the differences between a more traditional testing approach and the testing approach we want to embark on, we first will give a definition of a traditional testing approach in the context of our language.

Testing a component, in general, means to execute the component in order to increase confidence in its quality by inspecting the execution. In most cases, a component tester aims at only one feature to be tested at a time. Test cases are

specified each describing a specific execution of the program along with certain expectations related to the feature to be tested. In particular, *traditional* test case specifications basically consist of certain input data and the corresponding anticipated output data. Both, input and output data, may comprise not only values directly communicated between the component and its environment but also the component's states. For this reason, this testing approach is sometimes called *state-based testing* [28]. If the feature to be tested covers only functional requirements, the test case's pass and fail criteria are often completely confined to the specification of the input and output data: the component passes the specified test case if, and only if, its execution with respect to the specified input data leads to the desired output data. Moreover, the expected output data is often not given explicitly but instead a so-called *test oracle* is provided, i.e., a Boolean function of the input and output data determining whether the component passed or failed the test. This testing approach can be formalized in terms of our formal setting as follows:

Definition 2.5.2 (State-based testing)**:** For a given component under test p with $\Theta \vdash p : \Delta$, a state-based test case specification can be represented by

- input data $(h_{in}, \mathsf{v}_{in}, a_c)$ consisting of a heap function h_{in}, a global variable function v_{in}, and an incoming method call label $a_c = \nu(\Theta').\langle call\ o.m(\overline{v_{in}})\rangle?$ as well as of

- a test oracle function $success : (\mathsf{H} \times \mathsf{V} \times a) \rightarrow (a \times \mathsf{H} \times \mathsf{V}) \rightarrow \mathsf{bool}$ which yields for each input data a Boolean function of the resulting outgoing return label $a_r = \nu(\Delta').\langle return(v_{out})\rangle!$ and the final state of the component.

The component p passes the specified test case if:

$$\Delta \vdash (h_{in}, \mathsf{v}_{in}, \epsilon) : \Theta \stackrel{a_c a_r}{\Longrightarrow} \Delta' \vdash (h_{out}, \mathsf{v}_{out}, \epsilon) : \Theta'$$
$$\text{such that } success(h_{in}, \mathsf{v}_{in}, a_c)(a_r, h_{out}, \mathsf{v}_{out}) \text{ holds.}$$

In order to carry out a state-based test case execution, we have to write a test program which implements a *set-up*, assuring that the component's configuration corresponds to the specified input data, and which furthermore implements the method call a_c as well as the evaluation of the oracle function *success* afterwards.

Unfortunately, as the test program is itself a *Japl* component it can modify neither the global variables of the component under test nor its heap directly, but the test program has to drive the component to the desired input configuration by means of method calls. However, it is often not clear which calls should be made to reach the input configuration or if the configuration can be reached at all. For, it might be that there exists no trace in the component's trace semantics that leads to the desired configuration. Summarizing, the test set-up, initiating a state-based test execution, is difficult or sometimes even impossible to realize in a component-based setting, as the component's state is usually not accessible due to information hiding and encapsulation.

Besides that, according to Definition 2.5.2, state-based testing does not allow for call-backs which might occur between the specified method call and its return. More general, sometimes it might be necessary to specify a test execution which entails a longer communication sequence s in between the original call and its return:

$$\Delta \vdash (h_{in}, \mathsf{v}_{in}, \epsilon) : \Theta \overset{a_c\, s\, a_r}{\Longrightarrow} \Delta' \vdash (h_{out}, \mathsf{v}_{out}, \epsilon) : \Theta' \ .$$

Even if we relax the criteria for passing the test by allowing the component to produce more interface communication than merely the return label, then this requires at least additional declarations of the reaction that has to be carried out by the test program.

In a testing approach that is based on the component's interface behavior, in contrast, the component's internal state is not specified but only its behavior which is observable by a test program. This approach is therefore called *behavior-based testing*. In our setting, the observable behavior consists of method calls and returns occurring at the component's interface, thus, a test case specification stipulates a sequence of interface interactions.

Definition 2.5.3 (Behavior-based testing)**:** For a given component under test p with $\Theta \vdash p : \Delta$ a *behavior-based test case specification* can be represented by a sequence of communication labels, that is, a trace $s \in a^*$.

The component p passes the specified test case if:

$$\Delta \vdash c_{init}(p) : \Theta \overset{s}{\Longrightarrow} \Delta' \vdash c : \Theta'$$
$$\text{or } \Delta \vdash \overline{c_{init}}(p) : \Theta \overset{s}{\Longrightarrow} \Delta' \vdash c : \Theta' \ .$$

In other words, we actually test for $s \in [\![\Delta \vdash p : \Theta]\!]$. Note that this approach also allows for test cases which are initiated by the component under test, that is, we also can execute and test the component's main body. A test trace s generally describes communication steps which are expected to be carried out by the component under test as well as communication steps that have to be carried out by the component's environment. Thus, when conducting behavior-based testing, a test program need not to take care for establishing a specified input configuration of the component under test but in exchange it has to implement several calls and returns along the test execution as stipulated by the specified test trace. At the same time, the test program has to check that the component's contribution to the interface communication complies with the trace specification.

Specifically, in order to test for $s \in [\![\Delta \vdash p : \Theta]\!]$ for a component p and a test trace s, we need a test program which can implement the *complementary trace* $\bar{s}$ of s that results from s by replacing question marks with exclamation marks, and vice versa; hence, we require a test program p_{tp} with $\Theta \vdash p_{tp} : \Delta$ and $\bar{s} \in [\![\Theta \vdash p_{tp} : \Delta]\!]$. Then, a test execution can be considered as the execution of actually both programs p and p_{tp}, such that p_{pt}'s outgoing communication represents the incoming communication of p and vice versa such that p_{pt} reports a successful test run if it observes trace $\bar{s}$ at its interface. One question that may

arise from this testing approach is how to find an appropriate test program for a given test case specification. In fact, the following two chapters will deal with the introduction of, both, a test specification language that is built on the basis of interface traces and a test program generation algorithm.

But first it remains to show, that our language represents a sound framework for further investigations. More precisely, the semantics' extension by interface communication rules should represent a sound extension with regards to the semantics of the internal computation. In context of the above mentioned test approach, this means that, under the provision that the program p passes the test, the two programs p and p_{pt} can be merged to a single program that gives rise to a sequence of internal computation steps containing internal call and return steps that correspond to the labels of the trace s. This is more general formalized in the following lemma.

Definition 2.5.4 (Merge of components)**:** Let

$$p_1 = \overline{impdecl_1};\ \overline{T_1}\ \overline{x_1};\ \overline{cldef_1}\ \{\overline{T_1'}\ \overline{x_1'};\ stmt_1;\ \texttt{return}\}$$
$$\text{and}$$
$$p_2 = \overline{impdecl_2};\ \overline{T_2}\ \overline{x_2};\ \overline{cldef_2}\ \{\overline{T_2'}\ \overline{x_2'};\ stmt_2;\ \texttt{return}\}$$

be two components such that $\Theta_2 \vdash p_1 : \Theta_1$ and $\Theta_1 \vdash p_2 : \Theta_2$. Then the *merge of the two components* is defined as follows,

$$p_1 \Joinrel\Joinrel\Joinrel\

We annotated the transition arrows to indicate the context in which the respective transition rules are applied.

2. Assume a closed program p with $\vdash p : \Theta$. For every two components p_1 and p_2 with $\Theta_2 \vdash p_1 : \Theta_1$ and $\Theta_1 \vdash p_2 : \Theta_2$ such that $p_1 \Supset p_2 = p$ and $\Theta_1, \Theta_2 = \Theta$ the following holds:

   $c_{init}(p) \longrightarrow_p^* c$ implies the existence of a trace $t \in a^*$ such that:

   - $\Theta_2 \vdash c_{init}(p_1) : \Theta_1 \overset{t}{\Longrightarrow}_{p1} \Theta_2' \vdash c_1' : \Theta_1'$,
   - $\Theta_1 \vdash \overline{c_{init}}(p_2) : \Theta_2 \overset{\bar{t}}{\Longrightarrow}_{p2} \Theta_1' \vdash c_2' : \Theta_2'$ and
   - and $c_1' \Supset c_2' = c$.

In words: Feeding a component p_1 with the interface communication carried out by p_2, and vice versa, has the same effect as syntactically merging the two programs, such that the communication is carried out internally. On the other hand, a closed program can be torn into two components which can communicate via interface communication reaching configurations that correspond to the configurations reachable by the original component.

Chapter 3

The test specification language

In this chapter we will develop a test specification language which allows to specify unit tests for the *Java*-like programming language *Japl* introduced in the previous chapter. From the unit testing point of view, *Japl* components can be considered to be the smallest testable constituents of a *Japl* program, as one can test a *Japl* component without the need to modify its code but by providing a test program which imports the component and investigates its behaviour by means of method invocations. Therefore we will identify test units with *Japl* components. That is, a unit test always represents a test of a *Japl* component and in the following we will use the terms *unit* and *component* and, respectively, *unit test* and *component test* interchangeably.

A test specification shall stipulate a desired behavior to be shown by the unit under test to its environment. Equating units with components of our language, this means that such a specification talks about the communication which is captured by the communication labels of the external semantics introduced in Section 2.4.3. Thus, a first approach could be, to use the trace, i.e. the sequence of communication labels, itself as a test specification. Or, if one wants more expressiveness, also regular expressions of communication labels could form a test specification language. However, we want to ensure that the language provides some additional features which has an impact on the language design. In particular, we want that specifications of the language are:

interaction-based As mentioned above, the language shall allow for specifying the desired behavior of the unit in terms of the interactions that occur at the interface between the unit and its environment. More precisely, these interaction specifications represent the fundamental elements from the testing point of view and thus they deserve a correspondingly prominent role within the specification language from the language designing point of view.

executable The idea is to use a specification as a basis of an executable test program, or test driver, which, together with the component under test, actually performs the test that was specified. In other words, the test program has the task to determine whether the unit under test passes the specified test or not. As a consequence, a specification must not describe tests which cannot be implemented in the *Japl* programming language. This restriction has two aspects. On the one hand, a specification must not mention features of the unit under test which cannot be observed by a test program, which imports the unit as an external component and which therefore has no access to the internals of the unit. As an obvious example a specification must not include references to variables of the component under test, since its variables are not accessible, hence not observable, by the test program.

Another aspect of executability is given by the fact that in general a test specification does not only include observations but also stimuli of the unit under test. These stimuli are to be implemented by the test program and will show up in terms of interface communication during the test execution. Therefore, the sequence of stimuli and observations must comply to the control flow policy of the programming language. For instance, due to the sequential flow of control, a specification must not include two consecutive calls of a component method, as the test program cannot realize these method calls without accepting an incoming method call or return in between.

satisfiable While the executability criterion ensures the existence of a test program that can execute the specified test, satisfiability, in contrast, ensures that for each specified test a *Japl* component exists which can pass the test. For, in general, it is possible to write down sequences of interface interactions which could not be implemented by any component of our programming language. Let us consider again an example specification with two consecutive method calls — but this time let us assume that the specification requires the component under test to realize these calls. Again due to the sequential flow of control, no *Japl* component could fulfill such an expectation. The specification language should identify these *faulty specifications* and this should be performed preferably statically. In particular, syntax and type system should filter them out. Alas, in some cases this is not possible. For instance, a specification may require the evaluation of a Boolean expression to true. But in general it is impossible to decide statically whether a Boolean expression can be evaluated to true, at all. In Section 4.5 we will discuss in more detail that unfortunately there exist situations where we cannot even at runtime identify a specification as faulty. However, at least, we want to design our specification language such that we can single out as many faulty specifications as possible.

complete This is actually not a requirement regarding a single specification but rather concerns the specification language itself. That is, the language should

be complete in the sense that every interaction-based, executable, and satisfiable behavior should be expressible within the language.

accessible We want to encourage software developers to perform unit tests. Thus, software developers should be able to quickly learn the language. Moreover, testing should not break the rhythm of the short test-and-develop cycles which many programmers embark on due to extreme programming or other agile software development approaches.

It turns out that using traces, as defined in Definition 2.5.1, or regular expressions on communication labels do not meet most of the criteria. Indeed, not all sequences of labels described by a regular expression are satisfiable or executable. A trace, in contrast, satisfies most of the criteria by definition. However, a pure trace language is rather not accessible and in particular it is not very practical to use interaction traces as specifications since a trace does not entail any generalization but covers exactly only one specific behavior. Finally, it is difficult to define a specification language whose elements are only sequences of interactions which indeed represent a proper trace.

Our basic idea of meeting these requirements is to define a test specification language by *extending* the programming language with additional constructs that ease the specification of interactions. A specification represents a desired interaction trace (or a set of traces) to be shown by the unit under test. Extending the programming language means that developers only have to learn the additional constructs. Furthermore the design of the new constructs will exclude many faulty specifications on the syntax and type level already.

3.1 Extension by expectations

In Chapter 2 we have first defined a simple "monolithic" object oriented language which later has been extended to *Japl* by incorporating the notion of components. In this chapter we will in turn extend *Japl* in order to get a test specification language for testing *Japl* components. Again we will extend the original syntax and correspondingly extend and adapt the type system as well as the operational semantics. The formal definition of both, the *Japl* language and its extension, will allow for a formal definition of the test pass criteria and of the meaning of a test itself, too. For, an important consequence of our approach is that the extended operational semantics will provide a trace semantics for specifications similar to the trace semantics for *Japl* components defined in Section 2.5. Thus, it is natural to consider a specification's trace semantics to be the meaning of the test and it suggests itself to define the test pass criteria in terms of a relation regarding the test specification's trace semantics and the trace semantics of the component under test. Then, a strict and straightforward test pass criterion would be to demand trace inclusion: for each of the specification's traces there must exist a corresponding trace within the trace semantics of the component under test. Assuming that we use for the trace semantics of both, specifications and *Japl* components, the same notation $[\![\cdot]\!]$, we can also rephrase this test pass criterion

more formally by saying that a *Japl* component p satisfies a test specification s if the following holds:

$$[\![\Delta \vdash s : \Theta]\!] \subseteq [\![\Delta \vdash p : \Theta]\!]$$

Although the above formulated test pass criterion demonstrates the general idea of our approach, we will decide to slightly deviate from this relation in two aspects due to certain design decisions regarding the specification language.

First, a simple but crucial deviation comes from the fact that we will formalize specifications not from the point of view of the unit but of its environment. Thus, for instance, a call of a method of a unit class invoked by the unit's environment is expressed in a test specification in terms of an invocation of that method resulting in an *outgoing* call label within the external semantics of the test specification language. Within the trace semantics of the component under test, in contrast, this call shows up in form of an *incoming* call. The complementary viewpoints regarding the unit and the test specification resembles the situation of a programmer who is writing unit testing code for testing frameworks like *xUnit*.

Second, the detailed discussion about the extension below will show that our language will support relaxed specifications in that a specification may let the unit under test to chose from several admissible behaviors. For instance, instead of expecting exactly one specific incoming call[1] at a certain point of time, a specification may list several acceptable incoming calls. Providing alternatives regarding incoming communications likewise result in multiple traces within the semantics of the specification. A specification-conform component, however, needs only to realize one of these traces.

In the remainder of this section we will develop appropriate syntactical extensions of the test specification language along with an informal description of their meaning. For this, we will in particular account for the desired interaction-basedness and accessibility of the language. The subsequent three sections will then provide a formal definition of the syntax, the type system, and the operational semantics, respectively. Before we deal with the new constructs that we want to add, however, let us first see why it is necessary to define a specification language in the first place. That is, why is the original programming language *Japl* not expressive enough to formulate a specification (program) whose trace semantics can be used as a test specification for another component. According to the definition of the external semantics given in Table 2.12, we can identify the desired behavior of the unit as desired *incoming* communication steps of the external semantics. In particular, consider a *Japl* program p_s representing a test specification with

$$\Delta_0 \vdash c_{init}(p_s) : \Theta_0 \overset{t\gamma_1!}{\Longrightarrow} \Delta_1 \vdash c_1 : \Theta_1,$$

that is, the specification program is executed and produces a trace which ends with an outgoing communication label $\gamma_1!$. Now, the specification of the desired

[1] Note that we already use the *xUnit* perspective here. That is, the specified incoming call is to be implemented by the unit in terms of an outgoing call.

behavior could entail the fact that a certain incoming communication $\gamma_2?$ is expected to occur right after $\gamma_1!$:

$$\Delta_0 \vdash c_{init}(p_s) : \Theta_0 \overset{t\gamma_1!}{\Longrightarrow} \Delta_1 \vdash c_1 : \Theta_1 \xrightarrow{\gamma_2?} \Delta_2 \vdash c_2 : \Theta_2.$$

However, again according to the rules of the external semantics, the outgoing communication step represented by γ_1 either leads to an empty call stack or it puts a type-annotated receive statement on top of the call stack (CallO, NewO, and RetO). Thus, in the former case it is not determined whether the next incoming communication is an incoming method or constructor call (CallI, NewI) and in the latter case additionally an incoming return is possible (RetI). Moreover, the *Japl* program p_s has no influence on the input values, namely on the incoming return value or the input parameters of the call, respectively. We say a *Japl* program that has just given away the control to some external component is generally *input-enabled* meaning that it cannot decree a specific incoming communication to occur next but it accepts several different incoming calls and returns. This under-specification resulting from the openness of the program restrains us from stipulating the next *expected* incoming communication.

On account of this, we extend the language by *expectation statements* which determine the next expected incoming communication. This way, we will restrict the application of the semantics' incoming communication rules such that an application of a rule is only possible if the corresponding expectation statement is on top of the call stack.

Since we want to change the "look-and-feel" of the programming language as little as possible, the question arise how should these new statements look like and how to integrate them into the language. In order to specify incoming communication, we need statements for incoming method calls, incoming constructor calls, and incoming returns. The original programming language already provides statements for the outgoing counterparts: an outgoing method call is caused by a call statement, which includes the term $e.m(\overline{e})$, an outgoing constructor call includes the term `new` $C(\overline{e})$, and an outgoing return results from `return` e. It suggests itself that the terms for the incoming communication look similar. Thus, as a first approach we could, for instance, introduce a term for an incoming method call which resembles the conventional method call, except that it has a question mark instead of the usual dot for the method selector:

$$e?m(\overline{e}).$$

The usage of a question mark for expressing an incoming communication is inspired by *CSP* and other process calculi where a question mark describes the input of a value. An informal description of the term's semantics would be: wait for an incoming method call of method m of object e with actual parameters $\overline{e}$. However, sometimes we might want to give a more loose specification in that we don't want to stipulate the exact values of the actual parameters but only want to ensure that certain conditions for the values hold. It could be even the case

that we don't want to be specific regarding the callee object. As a consequence, the term for an incoming method call expectations has the following form:

$$(C\ x)?m(T_1\ x_1, \ldots, T_k\ x_k).\texttt{where}(e)\ .$$

The callee and parameter expressions in our first approach are now replaced by variable declarations which play the role of formal parameters expressing that the expectation is not specific regarding the incoming values. However, the new where-clause narrows down the possible incoming method calls, as the values of the parameters and of the callee must satisfy the condition e. Note, that a loose where-clause leads to many different possible incoming method calls, such that the specification does not only describe a single interaction trace (and its prefixes) anymore. However, it is certainly still possible to restrict the incoming communication to a distinct incoming method call by means of an appropriate where-clause which fixes the callee and incoming parameters to specific values, that is,

$$(C\ x)?m(T_1\ x_1, \ldots, T_k\ x_k).\texttt{where}(x{==}v\ \&\&\ x_1{==}v_1\ \&\&\ \ldots x_k{==}v_k)\ .$$

We add syntactic sugar for this kind of restrictions on incoming values, so that the last example can also be written as:

$$v?m(v_1, \ldots, v_k),$$

which resembles a usual method call a bit more, again. Moreover, it is allowed to omit the where-clause `where(true)`.

Similar to the terms for incoming method call expectations, we introduce terms for incoming constructor call and incoming return expectations, which are

$$\texttt{new}?(C\ x)C(\overline{T}\ x).\texttt{where}(e) \quad \text{and} \quad x = ?\texttt{return}(T\ x').\texttt{where}(e).$$

As in the case with incoming call specifications we likewise add syntactic sugar for incoming return terms. The term $?\texttt{return}(v)$ represents a shortform for $x = ?\texttt{return}(T\ x').\texttt{where}(x' == v)$ where x is a local variable which is not used somewhere else.

Using an extension of the programming language in order to specify test cases, may make the specification language more accessible for software developers. At the same time, it eases to satisfy the executability requirement, as we only have to ensure that the new statements can be translated to semantical equivalent program language code. All other statements can remain the same.

Moreover, it will become obvious that the specification language also meets the satisfiability requirement. For, the extension of the operational semantics will show, that we basically only introduce new premises in the incoming communication rules. Since we add only further restrictions it is easy to see that the extension of the language does not allow new traces that could not have been produced by a program of the original language already. However, adding restrictions could raise

the risk to produce faulty traces, that is, one could write specifications which could get stuck.

In particular, a specification gets stuck, if an incoming communication term represents an expectation which is inconsistent with the requirements for incoming communication that we introduced in Section 2.4.3. Fortunately, we can identify statically many of the specifications that would cause faulty traces. Specifically, we will explain in the following how we restrict the specification language, such that incoming communication expectations of a valid specification always comply with three of the four requirements, namely with well-typedness, control-flow consistency, and balance. The type system will ensure that a valid specification only contains expectations of incoming communication which is well-typed and consistent regarding the control flow. As for the balance requirement, we filter out undesired specification statically by introducing appropriate statements which incorporate the above mentioned expectation terms.

The balance condition stipulates that an incoming return may only occur if a corresponding outgoing call was processed previously. Since test specifications must not contain expectations that do not satisfy this requirement, we have to make sure that the term for incoming returns may only appear in certain situations. Remember, the argument for introducing the balance condition was that an outgoing return is always preceded by an incoming method call. This property in turn was due to the fact that return terms may only occur at the end of a method body, hence, it is actually caused by the syntactical structure of the code. The idea is to mirror the syntactical structure such that incoming return terms comply with the balance condition. More specifically, we define a new statement by combining the term for an (outgoing) method call with the corresponding incoming return, forming a dual version of a normal method definition, that is

$$e!m(\overline{e})\{\overline{T_l}\,\overline{x_l};\ \mathit{stmt};\ x = ?\texttt{return}(T\,x').\texttt{where}(e')\}.$$

Thus, the original statement of an (outgoing) method call, $x = e.m(\overline{e})$, is now split into the actual outgoing call and its corresponding incoming return such that the new construct indeed resembles a method definition: instead of a method signature we have an outgoing method call term and instead of the ususal return term we have an incoming return term. Combining the call and its return into one statement ensures that, assuming a syntactical valid specification, an incoming return term will never be executed without a preceding outgoing method call. At the same time the *expectation body* in between the outgoing call and the incoming return term makes it possible to define local variables $\overline{x_l}$ and a statement *stmt* in order to specify further interface interactions that are expected to happen in between the outgoing call and its return. Note, that we use the exclamation mark instead of a dot in the outgoing method call, which resembles the syntax of an output in *CSP*. Using an exclamation mark emphasis the duality to incoming method calls and makes the actual trace specification more explicit.

Using the same pattern we introduce a statement for the combination of an

outgoing constructor call and its return:

$$\texttt{new}!C(\overline{e})\{\overline{T_l}\,\overline{x_l};\ \mathit{stmt};\ x = ?\texttt{return}(C\ x').\texttt{where}(e')\}.$$

The remaining incoming communication terms that we still have to incorporate into our language are the incoming method and constructor call expectations. For similar reasons it again makes sense to use the same pattern, that is, to combine the incoming call term with a body that ends with an outgoing return. Thus, we introduce another statement that combines an incoming call expectation with an outgoing return:

$$(C\ x)?m(\overline{T}\,\overline{x}).\texttt{where}(e')\ \{\ \overline{T_l}\,\overline{x_l};\ \mathit{stmt};\ !\texttt{return}(e);\ \}$$

Finally, we define a similar statement for incoming constructor calls:

$$\texttt{new}(C\ x)?(\overline{T}\,\overline{x}).\texttt{where}(e')\ \{\ \overline{T_l}\,\overline{x_l};\ \mathit{stmt};\ !\texttt{return};\ \}$$

Note, in contrast to the incoming method call, the return term of an incoming constructor call does not include an expression for the returned value. For, a constructor always returns the name of the created object x.

Since we define all these constructs as statements, we can compose them in a nested and sequential way such that the resulting sequence of interface communication terms satisfies the balance condition.

Remark 3.1.1: Regarding incoming call statements, one could think that we actually do not need to introduce a completely new statement, since it might be sufficient to adapt the usual method definition. Indeed, an incoming call statement is almost identical to a method definition of a class — apart from the where-clause and the "formal parameter" for the callee object. But an incoming method call expectation is not only more specific regarding the expected values but, in contrast to a conventional method definition, it also is interpreted within a certain interaction context. More specifically, an incoming method call expectation deals with an incoming call resulting in a communication label which is expected to occur at a certain place within the interface trace while a conventional method definition is rather a template of a behavior shown by the method whenever it is called.

After this conceptional overview which also included an informal introduction of the interface communication statements, the following sections provide the details of the syntax, type system, and operational semantics of our test specification language.

3.2 Syntax

The syntax of the test specification language is given by a grammar as shown in Table 3.1. In general, the grammar of the test specification resembles that of the programming language given in Table 2.1 with small replacements and some extensions. To stress the extending character of the specification language the

$$
\begin{array}{rcll}
s & ::= & \overline{cutdecl}\ \overline{T}\ \overline{x};\ \overline{mokdecl}\ \{\ stmt\ \} & \text{specification} \\
cutdecl & ::= & \texttt{test class}\ C; & \text{test unit class} \\
mokdecl & ::= & \texttt{mock class}\ C\{C(T,\ldots,T);\ \overline{T\ m(T,\ldots,T);}\};\ & \text{mock class} \\
stmt & ::= & x = e \mid x = \texttt{new}\ C() \mid \varepsilon \mid stmt; stmt \mid \{\overline{T}\ \overline{x};\ stmt\} & \text{statements} \\
 & & \mid \texttt{while}\ (e)\ \{stmt\} \mid \texttt{if}\ (e)\ \{stmt\}\ \texttt{else}\ \{stmt\} & \\
 & & \mid\ stmt_{in}\ \mid\ stmt_{out}\ \mid\ \texttt{case}\ \{\ \overline{stmt_{in}; stmt}\ \} & \\
stmt_{in} & ::= & (C\ x)?m(\overline{T}\ \overline{x}).\texttt{where}(e)\ \{\overline{T}\ \overline{x};\ stmt;\ !\texttt{return}\ e\} & \text{incoming stmt} \\
 & & \mid \texttt{new}(C\ x)?C(\overline{T}\ \overline{x}).\texttt{where}(e)\ \{\overline{T}\ \overline{x};\ stmt;\ !\texttt{return}\} & \\
stmt_{out} & ::= & e!m(e,\ldots,e)\ \{\overline{T}\ \overline{x};\ stmt;\ x = ?\texttt{return}(T\ x).\texttt{where}(e)\ \} & \text{outgoing stmt} \\
 & & \mid \texttt{new}!C(e,\ldots,e)\ \{\overline{T}\ \overline{x};\ stmt;\ x = ?\texttt{return}(T\ x).\texttt{where}(e)\ \} & \\
e & ::= & x \mid \texttt{null} \mid \texttt{op}(e,\ldots,e) & \text{expressions}
\end{array}
$$

Table 3.1: Specification language for *Japl*: syntax

extensions are highlighted in the grammar definition. Similar to the original definition of a program p which consists of class import declarations, global variable definitions, class definitions, and a main body, the definition of a specification s consists of unit class declarations, global variable definitions, class declarations, and a specification body. In particular the class import declaration is replaced by the unit class declarations which also mention the names of the classes only. The class definition of a program is replaced by the *mock class* declaration where only the signature of the classes are specified. The method bodies are omitted since the specification body basically consists of the interaction trace and therefore implicitly stipulates the behavior of the classes, rendering the method body definitions unnecessary. As the classes do not provide method bodies or field declarations it wouldn't make sense to internally call their methods. Thus we omit the statements for (internal) method calls and field updates. For the same reason, the specification language only provides a simplified new construct which actually does not entail a constructor call but rather merely specifies the creation of a new object of a tester class. Furthermore, the specification language also provides sequential composition of statements, block statements, conditional statements, while loops, and the empty statement.

Finally, the language allows for explicitly specifying the interaction sequence between the tester program and the unit under test. To this end, we introduce dedicated statement for each type of interaction as discussed previously.

By introducing formal parameters in an incoming communication term we provided the possibility to relax a specification in terms of the expected incoming values. A case statement, where each branch consists of a sequence of statements which all start with an incoming call statement, enables a further relaxation with respect to the callee class and the called method or constructor, respectively: the tester's environment (i.e. the unit under test) chooses a branch by providing an incoming communication that matches the branch's leading incoming call statement.

Again, due to the lack of field declarations, we exclude field names from the set of possible expressions. Furthermore, due to the nested structure of the expectation specifications, the use of `this` would be ambiguous, hence is not supported. Instead, if we want to refer to the callee object of an incoming method call, we use the formal callee parameter of an incoming call term.

Remark 3.2.1: Although the intention of the specification language is to describe the interface interaction between an object-oriented component and its environment by specifying method calls, the specification language itself is *not* object-oriented. In particular, the language does not support the definition of (specification) classes but only the declaration of test class names and mock class signatures. An extension of the specification language with classes is discussed in Chapter 5.

3.3 Static semantics

Once we had developed useful syntactical constructs for specifying interface communication, defining the specification language's syntax was straightforward: basically, we just extended the statement definition of *Japl* by the new specification statements. In order to meet the language requirements from the beginning of this chapter, however, we have to further confine the valid specifications by means of the type system.

Recall, in particular, that executability requires a specified test to be implementable in terms of a *Japl* program and, respectively, satisfiability demands the existence of a *Japl* component which passes the test. With these requirements in mind consider the following specification snippet consisting of two nested outgoing method call statements:

$o_1!m_1(\overline{v_1})$ {
  $o_2!m_2(\overline{v_1})$ { ... }
  ...
};

Although this specification snippet represents a syntactical valid specification fragment, it must be considered as an invalid specification, as it cannot fulfill the executability requirement. For, as we have already pointed out, there exists no *Japl* program that implements the specified test: we cannot write a *Japl* program that realizes two consecutive outgoing method calls without an incoming communication in between, as the first outgoing method call passes the control to an other component rendering it impossible to invoke the second method call immediately afterwards. It is obvious that we can construct a dual example consisting of two nested incoming call statements which must be deemed an invalid specification too as it is not satisfiable.

The nested call statement example showed that we cannot use arbitrary statements as expectation body of a call statement, so considering an outgoing method call statement s_{out} with

$$s_{out} = o_1!m_1(\overline{v})\{\ \overline{T_l}\ \overline{x_l};\ \mathit{stmt}_1;\ x = ?\texttt{return}(T\ x').\texttt{where}(e)\ \},$$

as well as an incoming method call statement s_{in} with

$$s_{in} = o_2?m.\texttt{where}(e)(\overline{T}\ \overline{x})\{\ \overline{T_l}\ \overline{x_l};\ stmt_2;\ !\texttt{return}(v)\ \},$$

the question arises what kind of statements may be used for $stmt_1$ and, respectively, $stmt_2$ in general, in order to fulfill executability and satisfiability. To answer this question, it is important to understand the discrepancy between *Japl* and the specification language regarding their corresponding control flow policies. Due to the sequential flow of control, a *Japl* program is always blocked right after it has realized an outgoing communication and it may only proceed when the external semantics provides it with an incoming communication. This strict control flow policy does not hold for the specification language anymore. The above outgoing call statement s_{out}, in particular, indicates that a specification may proceed with the processing of $stmt_1$ after it has realized the outgoing call label $\langle call\ o_1.m_1(\overline{v})\rangle!$ due to the execution of the term $o_1!m_1(\overline{v})$. We refer to statements, like $stmt_1$, that occur between an outgoing communication and an incoming communication term as *passive statements* and we say they appear in *passive control context*. For, a *Japl* program that corresponds to the outgoing call statement gets blocked, hence it becomes *passive*, right after it has realized the outgoing call label $\langle call\ o_1.m_1(\overline{v})\rangle!$.

A *Japl* program that corresponds to the incoming call statement s_{in}, however, may proceed, i.e., it is *active*, right after it has realized the incoming call label $\langle call\ o_2.m_2(\overline{v})\rangle?$. Thus, we refer to statements, like $stmt_2$, occurring between an incoming and an outgoing communication term, as *active statements* and we say they appear in *active control context*.

Specifically, an incoming communication can "re-activate" a previously blocked *Japl* program again, hence, it is easy to see that we may use incoming call statements or the empty statement for passive statements, like $stmt_1$ in s_{out}, without breaking executability. In order to increase the expressiveness of our specification language, however, we will permit also other statements to appear in a passive control context. Consider, as an example, a specification where the expectation regarding incoming calls depends itself on an incoming value. More specifically, after performing an outgoing call $o!m(\overline{v})$, the specification expects a sequence of invocations of method m_1 of object o_1, where the exact number of invocations is determined by a Boolean input parameter x of method m_1. Then this can be expressed by the following specification snippet:

```
1  b = ...;
2  o!m() {
3    while(b) {
4      o1?m1(bool x) { b=x; ... }
5    }
6    o2?m2() { ... }
7    ...
8  };
```

The example demonstrates that the specification languages allows for a straightforward formalization of this kind of specifications. Note, however, it needs a

while-loop to appear in a passive control context therefore lacking a direct correspondent in *Japl*. Nevertheless, in Chapter 4 we will provide a code generation algorithm which allows to determine a *Japl* program that implements this test specification snippet. The algorithm's key concept for translating passive statements like the above mentioned passive while-loop is based on a *reordering* of the involved statements. Thus, we will allow statements to appear in passive control context if they do not entail side-effects as a reordering of these statements is not critical.

Considering the incoming call statement s_{in} with its active statement $stmt_2$ again, the situation is more relaxed, since in general $stmt_2$ can also be processed in *Japl*. As already mentioned above, the only exception is a statement which entails another incoming communication, because the sequential control flow of *Japl* does not allow two or more consecutive incoming call labels.

Be it as it may, regarding the typing system, it suffices to conclude that only incoming call statements, the empty statement, and side-effect-free statements, may appear in a passive control context. If a statement does not entail an incoming communication as the next interface communication, then it may appear in an active control context. This has to be checked by the type system.

The type system of the specification language is based on the type system of the *Japl* programming language which was introduced in Section 2.2 and Section 2.4.2. Recall, that in *Japl* well-typedness of a statement *stmt* was evaluated in context of a local type mapping Γ and a global type mapping Δ expressed by the typing judgment:

$$\Gamma; \Delta \vdash stmt : \mathsf{ok}.$$

As for the specification language we have to implement two modifications on the typing judgments for statements. First, we have to equip the typing judgments with an additional flag γ in order to implement the control-flow related checks that we have discussed above. The flag γ represents the considered control context of the statement and correspondingly ranges over the set $\{act, psv\}$.

Second, we have to ensure that a callee of an outgoing or incoming call statement indeed belongs to an external component or, respectively, to the program. To this end, we have to distinguish component and program classes in the typing judgments. In the *Japl* typing rules for statement, both, component and program classes, were included in the global type mapping Δ. Consequently, we split the global mapping into a global mapping Δ regarding component types and a global mapping Θ for program types.

Considering the two modifications, the specification language's type judgments for statements are of the following form:

$$\Gamma; \Delta; \Theta \vdash stmt : \mathsf{ok}^{\gamma}.$$

The type system of the specification language is given in Table 3.2. As mentioned earlier, it is based on the type system of *Japl*. Apart from the two modifications regarding the judgments, we introduce new rules for the new specification

$$[\text{T-Spec}]\ \frac{\Gamma;\Delta \vdash \overline{cutdecl} : \mathsf{ok} \quad \Theta = cltype(\overline{mokdecl}) \quad \Gamma' = \Gamma, \overline{x}{:}\overline{T} \quad \Gamma';\Delta;\Theta \vdash stmt : \mathsf{ok}^{\gamma};}{\Gamma;\Delta \vdash \overline{cutdecl}\ \overline{mokdecl}\ \overline{T}\ \overline{x};\ \{stmt;\ \texttt{return}\} : \Theta^{\gamma}}$$

$$[\text{T-CallIn}]\ \frac{\begin{array}{c}\Theta(C)(m).dom = \overline{T} \quad \Gamma, x{:}C, \overline{x}{:}\overline{T};\Delta,\Theta \vdash e : \mathsf{Bool} \quad \Theta(C)(m).ran = T \\ \Gamma' = \Gamma, x{:}C, \overline{x}{:}\overline{T}, \overline{x'}{:}\overline{T'} \quad \Gamma';\Delta;\Theta \vdash stmt : \mathsf{ok}^{act} \quad \Gamma';\Delta,\Theta \vdash e' : T\end{array}}{\Gamma;\Delta;\Theta \vdash (C\ x)?m(\overline{T}\ \overline{x}).\texttt{where}(e)\{\overline{T'}\ \overline{x'};\ stmt;\ !\texttt{return}\ e'\} : \mathsf{ok}^{psv}}$$

$$[\text{T-NewIn}]\ \frac{\begin{array}{c}\Theta(C)(C).dom = \overline{T} \quad \Gamma' = \Gamma, x{:}C, \overline{x}{:}\overline{T}, \overline{x'}{:}\overline{T'} \\ \Gamma';\Delta;\Theta \vdash stmt : \mathsf{ok}^{act} \quad \Gamma, x{:}C, \overline{x}{:}\overline{T};\Delta,\Theta \vdash e : \mathsf{Bool}\end{array}}{\Gamma;\Delta;\Theta \vdash \texttt{new}(C\ x)?C(\overline{T}\ \overline{x}).\texttt{where}(e)\{\overline{T'}\ \overline{x'};\ stmt;\ !\texttt{return}\} : \mathsf{ok}^{psv}}$$

$$[\text{T-CallOut}]\ \frac{\begin{array}{c}\Gamma;\Delta,\Theta \vdash e : C \quad \Gamma'(x) = \Delta(C)(m).ran \quad \Gamma;\Delta,\Theta \vdash \overline{e} : \Delta(C)(m).dom \\ \Gamma' = \Gamma, \overline{x}{:}\overline{T} \quad \Gamma';\Delta;\Theta \vdash stmt : \mathsf{ok}^{psv} \quad \Gamma';\Delta,\Theta \vdash e' : \mathsf{Bool}\end{array}}{\Gamma;\Delta;\Theta \vdash e!m(\overline{e})\{\overline{T}\ \overline{x};\ stmt;\ ?\texttt{return}(x).\texttt{where}(e')\} : \mathsf{ok}^{act}}$$

$$[\text{T-NewOut}]\ \frac{\begin{array}{c}\Gamma'(x) = C \quad \Gamma;\Delta \vdash \overline{e} : \Delta(C)(C).dom \\ \Gamma' = \Gamma, \overline{x}{:}\overline{T} \quad \Gamma';\Delta;\Theta \vdash stmt : \mathsf{ok}^{psv} \quad \Gamma';\Delta,\Theta \vdash e : \mathsf{Bool}\end{array}}{\Gamma;\Delta;\Theta \vdash \texttt{new}!C(\overline{e})\{\overline{T}\ \overline{x};\ stmt;\ ?\texttt{return}(x).\texttt{where}(e)\} : \mathsf{ok}^{act}}$$

$$[\text{T-VUpd}]\ \frac{\Gamma;\Delta,\Theta \vdash e : \Gamma(x)}{\Gamma;\Delta;\Theta \vdash x = e : \mathsf{ok}^{act}} \qquad [\text{T-Block}]\ \frac{\Gamma, \overline{x}{:}\overline{T};\Delta;\Theta \vdash stmt : \mathsf{ok}^{act}}{\Gamma;\Delta;\Theta \vdash \{\overline{T}\ \overline{x};\ stmt\} : \mathsf{ok}^{act}}$$

$$[\text{T-NewInt}]\ \frac{C \in dom(\Theta) \quad \Gamma(x) = C}{\Gamma;\Delta;\Theta \vdash x = \texttt{new}\ C() : \mathsf{ok}^{act}}$$

$$[\text{T-Seq}]\ \frac{x\Gamma;\Delta;\Theta \vdash stmt_1 : \mathsf{ok}^{\gamma} \quad \Gamma;\Delta;\Theta \vdash stmt_2 : \mathsf{ok}^{\gamma}}{\Gamma;\Delta;\Theta \vdash stmt_1;\ stmt_2 : \mathsf{ok}^{\gamma}}$$

$$[\text{T-While}]\ \frac{\Gamma;\Delta;\Theta \vdash e : \mathsf{Bool} \quad \Gamma;\Delta;\Theta \vdash stmt : \mathsf{ok}^{\gamma}}{\Gamma;\Delta;\Theta \vdash \texttt{while}\ (e)\ \{stmt\} : \mathsf{ok}^{\gamma}}$$

$$[\text{T-Cond}]\ \frac{\Gamma;\Delta;\Theta \vdash e : \mathsf{Bool} \quad \Gamma;\Delta;\Theta \vdash stmt_1 : \mathsf{ok}^{\gamma} \quad \Gamma;\Delta;\Theta \vdash stmt_2 : \mathsf{ok}^{\gamma}}{\Gamma;\Delta;\Theta \vdash \texttt{if}\ (e)\ \{stmt_1\}\ \texttt{else}\ \{stmt_2\} : \mathsf{ok}^{\gamma}}$$

$$[\text{T-Case}]\ \frac{\Gamma;\Delta;\Theta \vdash \overline{stmt_{in}} : \mathsf{ok}^{psv} \quad \Gamma;\Delta;\Theta \vdash \overline{stmt} : \mathsf{ok}^{psv}}{\Gamma;\Delta;\Theta \vdash \texttt{case}\ \{\ \overline{stmt_{in}; stmt}\ \} : \mathsf{ok}^{psv}}$$

Table 3.2: Specification language for *Japl*: type system (stmts)

statements and we skip the rules that deal with class definitions and other omitted constructs of the original language. A specification is type-checked by using rule T-Spec. The rule determines the committed type context Θ by extracting the class types from the specification's mock class signatures. Moreover, it checks if

the classes of the component under test are among the types of the assumed type context Δ. Finally, it type-checks the body statement within a typing context that is given by the assumption context, the commitment context, as well as the local context enriched by the global variables. The type-check of the body statement yields a control context γ which is also used to annotated the committed types of the specification, indicating that the specified test starts in passive or in active control context, respectively.

The rules T-CallIn and T-NewIn deal with the incoming method and constructor call statements and resemble the now unnecessary rule T-MDef for method definitions of Table 2.2. After extending the local type context with the "formal parameters", the local variables, and the callee object, we have to type-check the body statement and, in case of a method call, the return expression. Moreover, a call statement is only well-typed if it appears in a passive control context and if the callee class C is an element of the program context Θ. Right after the incoming call, the program has gained control and thus the body statement is correspondingly checked in an active control context.

In a similar way, outgoing method and constructor call statements may only appear in a situation where the program has the control which is again ensured by an exclamation mark in the context of the judgment that forms the conclusion. Thus, the body statement in turn has to be checked in a passive control context.

We want to allow sequential composition of incoming call statements. Therefore, the rule T-Seq can be applied in an active as well as in a passive control context. This is done by using a variable γ for the control context. However, both sub-statements have to be well-typed regarding the same control context. We similarly proceed with while loops (T-While) and conditional statements (T-Cond). Allowing the latter two kind of statements to appear in a passive control context considerably increases the expressiveness of the specification language, as we have shown already. However, the rules T-Block and T-VUpd show that we allow block variable declarations and assignments in an active control context only, because they involve a side-effect. The case statement is only well-typed in a passive control context and also all its sub-statements have to be well-typed in a passive control context.

Finally, we have to carry out minor adaptions to transform the rule T-Prog' for open programs of Table 2.9 to T-Spec for specifications. The import declaration check of rule T-Prog' is replaced by a unit declaration check. Furthermore, the function *cltype* has to be adapted, as the mock class declarations consist only of the method signatures but do not provide method bodies.

Definition 3.3.1 (Well-typedness)**:** A specification s is *well-typed* if there exist an assumption/commitment context Δ, Θ and a control context γ such that the judgment

$$; \Delta \vdash s : \Theta^{\gamma}$$

is deducible by means of the deduction rules given in Table 3.2 and 2.3. In particular, the deduction starts with an empty local type mapping. Therefore, well-typedness of the

specification s is denoted by

$$\Delta \vdash s : \Theta^{\gamma}.$$

However, sometimes we will omit the control context annotation meaning that s is well-typed either in a passive or in an active control context.

Note, although some statements can in general occur in a passive or in an active control context, they are always well-typed within either a passive or an active control context, only, depending on the code context. If, for instance, a conditional statement forms the body of an outgoing call statement, then it is well-typed in a passive control context. If, in contrast, it forms the body of an incoming call statement, then it appears in an active control context.

Remark 3.3.2: Incoming call statements are well-typed in passive control context, only. Their bodies in turn are only well-typed in active control context. The dual holds for outgoing call statements. Together with the nested nature of the call statements, this leads always to executions with interaction sequences that are consistent regarding the control-flow at the interface.

The grammar given in Table 3.2 was motivated to show that the specification language indeed represents basically a simple extension of the programming language. Due to the relaxed control flow policy of the specification language, however, we had to add some extra checks within the type system in order to ensure executability and satisfiability. Specifically, we added the notion of active and passive control contexts as well as active and passive statements. It is also possible to implement the control-flow related checks in the syntax definition already. In particular, we can distinguish active and passive statements on the syntax level. For this, consider the following definition.

Definition 3.3.3 (Active and passive statements: s^{act}, s^{psv})**:** The syntax for active and passive statements, s^{act} and s^{psv}, respectively, is given in terms of the following grammar where e refers to expressions as defined in Table 3.2:

$$\begin{array}{rcl}
s^{psv} & ::= & \texttt{if}(e)\ \{s^{psv}\}\ \texttt{else}\ \{s^{psv}\} \mid \texttt{while}(e)\ \{s^{psv}\} \mid s^{psv};\ s^{psv} \\
 & & \mid stmt'_{in} \mid \texttt{case}\ stmt'_{in}; s^{psv} \\
stmt'_{in} & ::= & (C\ x)?m(\overline{T}\ \overline{x}).\texttt{where}(e)\ \{\overline{T}\ \overline{x};\ s^{act};\ !\texttt{return}\ e\} \\
 & & \mid \texttt{new}(C\ x)?C(\overline{T}\ \overline{x}).\texttt{where}(e)\ \{\overline{T}\ \overline{x};\ s^{act};\ !\texttt{return}\} \\
s^{act} & ::= & \texttt{if}(e)\ \{s^{act}\}\ \texttt{else}\ \{s^{act}\} \mid \texttt{while}(e)\ \{s^{act}\} \mid s^{act};\ s^{act} \\
 & & \mid x = e \mid \{\overline{T}\ \overline{x};\ s^{act}\} \mid stmt'_{out} \\
stmt'_{out} & ::= & e!m(e,\ldots,e)\ \{\overline{T}\ \overline{x};\ s^{psv};\ x = ?\texttt{return}(T\ x).\texttt{where}(e)\ \} \\
 & & \mid \texttt{new}!C(e,\ldots,e)\ \{\overline{T}\ \overline{x};\ s^{psv};\ x = ?\texttt{return}(T\ x).\texttt{where}(e)\ \}.
\end{array}$$

The fact that conditional statements, while-loops, and sequential compositions may appear in active and in passive control context is reflected within Definition 3.3.3, in that parts of the original definition of *stmt* are duplicated to corresponding parts in s^{psv} and s^{act}. The side-effect entailing assignments and

block statements, however, are always instances of s^{act}. Note that we also had to redefine the syntax definition for the incoming and outgoing communication statements, as the new versions, $stmt'_{out}$ and $stmt'_{in}$, account for the control-flow policy. That is, an incoming call statement now has always an active statement as expectation body and an outgoing call statement a passive statement.

The following lemma will relate the syntax definition for active and passive statements with the original definition of the specification language. More specifically, the lemma will show that all statements within a syntactical valid and well-typed specification are instances of either s^{act} or s^{psv}.

Lemma 3.3.4: Let

$$s = \overline{cutdecl}\ \overline{T}\ \overline{x};\ \overline{mokdecl}\ \{\ stmt\ \}$$

be a well-typed specification such that $\Delta \vdash s : \Theta$. Then $stmt$ is either of the form s^{act} or of the form s^{psv}.

Proof. By structural induction. We show that $\Gamma; \Delta; \Theta \vdash stmt : \mathsf{ok}^{psv}$ implies that $stmt$ is of the form s^{psv} and that $\Gamma; \Delta; \Theta \vdash stmt : \mathsf{ok}^{act}$ implies that $stmt$ is of the form s^{act}. We show some cases regarding the form of $stmt$:

Case $(C\ x)?m(\overline{T}\ \overline{x}).\texttt{where}(e)\{\overline{T'}\ \overline{x'};\ stmt;\ !\texttt{return}\ e'\}$
Well-typedness of s yields

$$\Gamma; \Delta; \Theta \vdash (C\ x)?m(\overline{T}\ \overline{x}).\texttt{where}(e)\{\overline{T'}\ \overline{x'};\ stmt';\ !\texttt{return}\ e'\} : \mathsf{ok}^{psv}$$

and thus $\Gamma; \Delta; \Theta \vdash stmt' : \mathsf{ok}^{act}$. Due to the induction hypothesis we know that $stmt'$ is of the form s^{act}. And this in turn implies that $stmt$ is of the form s^{psv}.

Case $x = e$
We know that $\Gamma; \Delta; \Theta \vdash x = e : \mathsf{ok}^{act}$. Moreover, $x = e$ is an instance of s^{act}.

Case $stmt_1; stmt_2$

Subcase
Assume $\Gamma; \Delta; \Theta \vdash stmt_1;\ stmt_2 : \mathsf{ok}^{act}$. Then also $\Gamma; \Delta; \Theta \vdash stmt_1 : \mathsf{ok}^{act}$ and $\Gamma; \Delta; \Theta \vdash stmt_2 : \mathsf{ok}^{act}$. The induction hypothesis yields that both, $stmt_1$ and $stmt_2$, are of the form s^{act}. Thus, also the sequence is an instance of s^{act}.

Subcase A
ssume $\Gamma; \Delta; \Theta \vdash stmt_1;\ stmt_2 : \mathsf{ok}^{psv}$. Then also $\Gamma; \Delta; \Theta \vdash stmt_1 : \mathsf{ok}^{psv}$ and $\Gamma; \Delta; \Theta \vdash stmt_2 : \mathsf{ok}^{psv}$. The induction hypothesis yields that both, $stmt_1$ and $stmt_2$, are of the form s^{psv}. Thus, also the sequence is an instance of s^{psv}. □

Assuming a well-typed specification, it is often more convenient to use the syntax definition for active and passive statements instead of the general statement definition $stmt$ within proofs and definitions. In particular, we will use s^{act} and s^{psv} in the following section which deals with the definition of the operational semantics.

3.4 Operational semantics

In general the operational semantics of the specification language is very similar to the operational semantics of the original programming language. In particular, the internal steps remain the same. Regarding the inference rules of the external steps, the crucial point is that we have to narrow down the communication steps such that the resulting trace semantics of the specification consists only of the specified traces (and their prefixes). This is implemented, on the one hand, by additional premises and, on the other hand, by allowing incoming communication only if a corresponding communication term is on top of the call stack.

The different handling of interface communication as well as the absence of internal method and constructor calls also leads to a somewhat different, i.e., simpler, form of the call stack of a specification. For, the execution of a program never adds or removes an activation record but each inference rule only modifies the topmost activation record. Although this means that the call stack does not consist of several blocked and possibly one active activation record, we still distinguish activation records which only allow incoming communication as the next interface communication from activation records which only allow outgoing communication as the next interface communication. Thus, for the activation records of the specification language we define

$$
\begin{array}{rcl}
\mathsf{AR} & ::= & \mathsf{AR}^a \mid \mathsf{AR}^p \\
\mathsf{AR}^a & ::= & (\mu, mc^{act}) \\
\mathsf{AR}^p & ::= & (\mu, mc^{psv}) \\
mc^{act} & ::= & s^{act} \mid s^{act};\ !\texttt{return}(e);\ mc^{psv} \\
mc^{psv} & ::= & s^{psv} \mid s^{psv};\ x = ?\texttt{return}(T\ x).\texttt{where}(e);\ mc^{act}
\end{array}
$$

The rules of the operational semantics are given in Table 3.3.

The rules CallO and NewO deal with outgoing method and, respectively, constructor call statements. Just as the corresponding rules of the programming language, the expressions within the actual call term are evaluated and the transition is labeled with an outgoing call label. However, in the resulting configuration, the call stack is not blocked by a receive statement but instead only the actual call term of the statement is removed leaving the body of the call statement on top of the call stack. For, the body of the call statement comprises the desired tester/environment interactions that should occur until the call's incoming return occurs. The variable structure is extended by a variable function for the local variables of the call statement. Note that, although CallO and NewO resemble the corresponding rules of the programming language we do not add an activation record as we did in the semantics of the programming language. Otherwise the local variables of this call statement wouldn't be accessible by the body statement. Finally, the return statement is annotated with the return type of the called method or, respectively, the callee's class name.

$$[\textsc{CallO}]\ \frac{\begin{array}{c} s^{act} = e!m(\overline{e})\ \{\overline{T}\ \overline{x};\ s^{psv};\ x = ?\texttt{return}(T\ x').\texttt{where}(e')\} \\ a = \nu(\Theta').\langle call\ o.m(\overline{v})\rangle!\quad o \in dom(\Delta) \end{array}}{\begin{array}{c} \Delta \vdash (h, \mathsf{v}, (\mu, s^{act}; mc^{act}) \circ \mathsf{CS}) : \Theta \xrightarrow{a} \\ \Delta \vdash (h, \mathsf{v}, (\mathsf{v}_l \cdot \mu, s^{psv}; x = ?\texttt{return}(T\ x').\texttt{where}(e'); mc^{act}) \circ \mathsf{CS}) : \Theta, \Theta' \end{array}}$$

where $o = [\![e]\!]_h^{\mathsf{v},\mu}$, $\overline{v} = [\![\overline{e}]\!]_h^{\mathsf{v},\mu}$, $T = \Delta^2(o)(m).ran$, $\Theta' = \mathsf{new}(h, \overline{v}, \Theta)$, and $\mathsf{v}_l = \{\overline{x} \mapsto ival(\overline{T})\}$

$$[\textsc{NewO}]\ \frac{\begin{array}{c} s^{act} = \texttt{new}!C(\overline{e})\ \{\overline{T}\ \overline{x};\ s^{psv};\ x = ?\texttt{return}(C\ x').\texttt{where}(e')\} \\ a = \nu(\Theta').\langle new\ C(\overline{v})\rangle!\quad C \in dom(\Delta) \end{array}}{\begin{array}{c} \Delta \vdash (h, \mathsf{v}, (\mu, s^{act}; mc^{act}) \circ \mathsf{CS}) : \Theta \xrightarrow{a} \\ \Delta \vdash (h, \mathsf{v}, (\mathsf{v}_l \cdot \mu, s^{psv}; x = ?\texttt{return}(C\ x').\texttt{where}(e'); mc^{act} \circ \mathsf{CS}) : \Theta, \Theta' \end{array}}$$

where $\overline{v} = [\![\overline{e}]\!]_h^{\mathsf{v},\mu}$, $\Theta' = \mathsf{new}(h, \overline{v}, \Theta)$, and $\mathsf{v}_l = \{\overline{x} \mapsto ival(\overline{T})\}$

$$[\textsc{RetO}]\ \frac{\begin{array}{c} a = \nu(\Theta').\langle return(v)\rangle! \end{array}}{\begin{array}{c} \Delta \vdash (h, \mathsf{v}, (\mathsf{v}_l \cdot \mu, !\texttt{return}\ e;\ mc^{act}) \circ \mathsf{CS}) : \Theta \xrightarrow{a} \\ \Delta \vdash (h, \mathsf{v}, (\mu, mc^{act}) \circ \mathsf{CS}) : \Theta, \Theta' \end{array}}$$

where $v = [\![e]\!]_h^{\mathsf{v},\mathsf{v}_l \cdot \mu}$ and $\Theta' = \mathsf{new}(h, v, \Theta)$

$$[\textsc{CallI}]\ \frac{\begin{array}{c} s^{psv} = (C\ x)?m(\overline{T}\ \overline{x}).\texttt{where}(e')\ \{\overline{T_l}\ \overline{x_l};\ s^{act};\ !\texttt{return}\ e\} \\ a = \nu(\Delta').\langle call\ o.m(\overline{v})\rangle?\quad C = \Theta(o)\quad \Theta \vdash a : \Delta \\ [\![e']\!]_h^{\mathsf{v},\mathsf{v}_l \cdot \mu} = true \end{array}}{\begin{array}{c} \Delta \vdash (h, \mathsf{v}, (\mu, s^{psv};\ mc^{psv}) \circ \mathsf{CS}) : \Theta \xrightarrow{a} \\ \Delta, \Delta' \vdash (h, \mathsf{v}, (\mathsf{v}_l \cdot \mu, s^{act};\ !\texttt{return}\ e;\ mc^{psv} \circ \mathsf{CS}) : \Theta \end{array}}$$

where $\mathsf{v}_l = \{x \mapsto o, \overline{x} \mapsto \overline{v}, \overline{x_l} \mapsto ival(\overline{T_l})\}$

$$[\textsc{NewI}]\ \frac{\begin{array}{c} s^{psv} = \texttt{new}?(C\ x)C(\overline{T}\ \overline{x}).\texttt{where}(e')\ \{\overline{T_l}\ \overline{x_l};\ s^{act};\ !\texttt{return}\} \\ a = \nu(\Delta').\langle new\ C(\overline{v})\rangle?\quad C \in dom(\Theta)\quad \Theta \vdash a : \Delta \\ [\![e]\!]_h^{\mathsf{v},\mathsf{v}_l \cdot \mu} = true \end{array}}{\begin{array}{c} \Delta \vdash (h, \mathsf{v}, (\mu, s^{act};\ mc^{act}) \circ \mathsf{CS}) : \Theta \xrightarrow{a} \\ \Delta, \Delta' \vdash (h', \mathsf{v}, (\mathsf{v}_l \cdot \mu, s^{act};\ !\texttt{return}\ x;\ mc^{act}) \circ \mathsf{CS}) : \Theta \end{array}}$$

where $o \in N \setminus dom(h)$, $h' = h[o \mapsto Obj_{\perp}^{C}]$, and $\mathsf{v}_l = \{x \mapsto o, \overline{x} \mapsto \overline{v}, \overline{x_l} \mapsto ival(\overline{T_l})\}$

$$[\textsc{RetI}]\ \frac{\begin{array}{c} a = \nu(\Delta').\langle return(v)\rangle?\quad \Delta \vdash a : \Theta\quad \Delta, \Delta', \Theta \vdash v{:}T \\ [\![e]\!]_h^{\mathsf{v},\{x' \mapsto v\} \cdot \mathsf{v}_l \cdot \mu} = true \end{array}}{\begin{array}{c} \Delta \vdash (h, \mathsf{v}, \mathsf{v}_l \cdot \mu, x = ?\texttt{return}(T\ x').\texttt{where}(e); mc^{act} \circ \mathsf{CS}) : \Theta \xrightarrow{a} \\ \Delta, \Delta' \vdash (h, \mathsf{v}', (\mu', mc^{act}) \circ \mathsf{CS}) : \Theta \end{array}}$$

where $(\mathsf{v}', \mathsf{v}_l' \cdot \mu') = vupd(\mathsf{v}, \mathsf{v}_l \cdot \mu, x \mapsto v)$

$$[\textsc{CaseI}_C]\ \frac{\begin{array}{c} stmt_{in} = (C\ x)?m(\overline{T}\ \overline{x}).\texttt{where}(e')\ \{\overline{T_l}\ \overline{x_l};\ s^{act};\ !\texttt{return}\ e\} \\ a = \nu(\Delta').\langle call\ o.m(\overline{v})\rangle?\quad C = \Theta(o)\quad \Theta \vdash a : \Delta \\ [\![e']\!]_h^{\mathsf{v},\mathsf{v}_l \cdot \mu} = true \end{array}}{\begin{array}{c} \Delta \vdash (h, \mathsf{v}, (\mu, \texttt{case}\ \{\overline{stmt}\};\ mc^{psv}) \circ \mathsf{CS}) : \Theta \xrightarrow{a} \\ \Delta, \Delta' \vdash (h, \mathsf{v}, (\mathsf{v}_l \cdot \mu, s^{act};\ !\texttt{return}\ e;\ stmt';\ mc^{psv} \circ \mathsf{CS}) : \Theta \end{array}}$$

where $stmt_{in}; stmt' \in \overline{stmt}$, $\overline{T}\ \overline{x} = mparams(C, m)$, and $\mathsf{v}_l = \{x \mapsto o, \overline{x} \mapsto \overline{v}, \overline{x_l} \mapsto ival(\overline{T_l})\}$

$$[\textsc{CaseI}_N]\ \frac{\begin{array}{c} stmt_{in} = \texttt{new}?(C\ x)C(\overline{T}\ \overline{x}).\texttt{where}(e')\ \{\overline{T_l}\ \overline{x_l};\ s^{act};\ !\texttt{return}\} \\ a = \nu(\Delta').\langle new\ C(\overline{v})\rangle?\quad C \in dom(\Theta)\quad \Theta \vdash a : \Delta \\ [\![e]\!]_h^{\mathsf{v},\mathsf{v}_l \cdot \mu} = true \end{array}}{\begin{array}{c} \Delta \vdash (h, \mathsf{v}, (\mu, \texttt{case}\ \{\overline{stmt}\};\ mc^{psv}) \circ \mathsf{CS}) : \Theta \xrightarrow{a} \\ \Delta, \Delta' \vdash (h', \mathsf{v}, (\mathsf{v}_l \cdot \mu, s^{act};\ !\texttt{return}\ x;\ mc^{act}) \circ \mathsf{CS}) : \Theta \end{array}}$$

where $stmt_{in}; stmt \in \overline{stmt}$, $\overline{T}\ \overline{x} = mparams(C, m)$, and $\mathsf{v} = \{x \mapsto o, \overline{x} \mapsto \overline{v}, \overline{x_l} \mapsto ival(\overline{T_l})\}$

Table 3.3: Specification language for *Japl*: operational semantics (external)

The rule RetO is almost identical to the former version, except that we do not have to remove an activation record from the call stack. Likewise, we only remove a variable function but not a method variable structure.

The rules CallI and NewI can only be applied if the statement on top of the stack frame is indeed an incoming method call statement or an incoming constructor call statement, respectively. Additionally, we add a premise which asserts that the where-clause condition evaluates to true. The evaluation uses a variable context which is already extended by the formal parameters of the call terms, as the where-clause expression might contain references to parameters. Again, only the call term is removed from the call stack.

Rule RetI deals with the incoming return term, which has been annotated with the proper return type. After the transition, the variable context is shortened by the top most variable function, since it represented the variables of the call statement which the return term belonged to. Note, that we first updated the old variable context with the incoming return value since we do not know whether the target variable was part of the call statement's variables.

The last rules CaseI_C and CaseI_N deal with the case statement. These rules are applicable exactly if rule CallI or rule NewI is applicable for at least one of its branches. One might think, it would be more straightforward to provide an internal rule which just reduces the case statement non-deterministically to one of its branches. However, not the specification but the external component should non-deterministically choose a branch.

Leaving a statement on top of the stack frame after an outgoing call term has been processed, results in a crucial change of the language. Right after the call, the program is not blocked waiting for an incoming communication but it still can proceed. Although the type system ensures that assignments may not occur right after an outgoing call, still while-loops and conditional statements may be processed by means of *internal* communication steps. Thus, regarding internal computation steps, the specification language breaks the control flow requirement here. However, concerning the *interface* communication, also a specification still sticks to this requirement. For, the typing rules do not allow a nesting of statement which results in two consecutive incoming or two consecutive outgoing communication terms. As a consequence, the traces of a specification program always satisfy the control flow requirement.

Allowing while-loops and conditional statements in a passive control context, however, eases the definition of trace-based specifications. A while-loop in a passive control context allows to specify repetitions of incoming calls where the exact number of repetitions depends on the incoming values and is, thus, not known statically. Conditional statements in a passive control context allow to specify different expectations depending on conditions unknown statically.

Remark 3.4.1: Note, the lack of class definitions implies that all transition rules do not depend on the specification code. That is, we do not have to index transition steps by a specification.

Finally, we give a definition for test specification executions and traces that corresponds to Definition 2.4.8. Moreover, we expand the trace semantics definition given in Definition 2.5.1 in order to include specifications.

Definition 3.4.2 (Specification execution; specification traces)**:** Let

$$s \equiv \overline{cutdecl}\ \overline{T}\ \overline{x};\ \overline{mokdecl}\ \{stmt;\ \texttt{return}\}$$

be a specification with $\Delta \vdash s : \Theta$. We, again, broaden the application of c_{init}, defined in Definition 2.3.3 and Definition 2.4.8, such that we also apply it to specifications s.

The execution of a specification is represented by a finite, possibly empty, sequence of internal and external transitions starting from its initial configuration. The sequence of communication labels arising from an execution is called an *(observable interaction) trace* of the specification. As in the case of program executions, we use an annotated arrow $\overset{t}{\Longrightarrow}$ to represent a specification execution that implements the trace t. The corresponding rules are given in Table 2.13. Thus, the execution of a specification represents the reflexive transitive closure of the internal and external transitions.

Note, that due to the relaxed control-flow policy, specifications do not have passive but only active initial configurations. For, not a passive initial configuration but a passive main statement is used to express a specification execution that starts with an incoming communication. Similarly, the following definition for the trace semantics of specifications gets by with only one semantic function.

Definition 3.4.3 (Trace Semantics)**:** We expand the domain of the semantic function $[\![\cdot]\!]$, given in Definition 2.5.1, to $\Delta \vdash s : \Theta$, where s represents a well-typed specification. In particular, assuming $\Delta \vdash s : \Theta$, we define

$$[\![\Delta \vdash s : \Theta]\!] \stackrel{\text{def}}{=} \{s \in a^* \mid \Delta \vdash c_{init}(s) : \Theta \overset{s}{\Longrightarrow} \Delta' \vdash c' : \Theta'\}$$

3.5 Example

Having defined the test specification language, let us have a look at two small example specifications. The first example specifies the proper usage of a simple file system library. The second example represents a test specification for the voter system introduced in the Chapter 1.

Consider a system's library for handling files equipped with a specific application programmer's interface (API). Usually, such an API entails a reasonable orders of file operations that may be invoked by a program. In particular, let us assume that the operations for writing strings to a file consists of an open-for-writing operation, a sequence of write-string operations, and a final close operations. Further, let us assume that a class *File* encapsulates these operations, such that they are accessible via method calls. The class' constructor is equipped with a string parameter for specifying the file name. A method *openWrite* allows to request for opening the file for writing. The method's Boolean return value indicates a successful or a failed execution of the file operation. Further, the class provides a method *writeStr* which writes its string parameter to the corresponding file. It returns the string that actually has been written to the file (which could be,

for instance, a prefix of the method's parameter due to the lack of disk space). Finally, an invocation of the *close* method closes the file and possibly allows to release system resources that were used for handling the file. Again, a successful execution of the underlying file close operation is committed with `true`.

The valid order of file-writing operations therefore corresponds to a sequence of constructor and method calls regarding class *File*. Calling the method *writeStr* before the file has been opened via *openWrite*, for instance, doesn't make sense. Instead, ignoring the method returns, the valid sequences can be depicted by the following graph:

where we assume f to be the object that has been created by the constructor call at the beginning of the sequence.

Knowing about the interface of *File* and the valid method invocation order, we can specify the behavior to be shown by a program that uses *File* for writing string files. The corresponding example specification is given in Listing 3.1.

The specification starts with the declaration of global variables. Since the specification does not contain callbacks to the component under test we do not need to specify its class names, hence we dropped the test class declaration. Lines 4 to 8 deal with the interface declaration of the *File* class as described above.

While the interface declaration stipulates the static aspects of the interface, the behavior specification given in Lines 10 to 30 deal with its dynamic aspects. Lines 11 to 14 represent the expectation of an incoming constructor call: a file-write task always starts with the creation of a *File* object. The expectation's where-clause checks whether the string represents a valid file name. Here, we just ensure that the parameter is not the empty string. We store the name of the created object f in the global variable *file* as the scope of the constructor call expectation ends in Line 14.

After the object creation, we expect the object's *openWrite* method to be called. This is expressed in Line 15 to 18. Within the incoming call term in Line 15, we use the global variable *file* to ensure that indeed exactly the object is called that just has been created.[2] The actual file-operation for opening the file is replaced by an assignment to the global Boolean variable *writing*. It encodes the file state regarding write operations, in that the value `true` indicates the file's readiness for writing. Correspondingly, the value is passed as the return value to the component under test.

Once the file is ready for writing, the component under test may write an arbitrary number of strings to the file until it finally calls *close* to close it. This is implemented in terms of a passive while-loop in Line 19 to 29. As long as *writing* is `true`, the component under test is allowed to call the method *writeStr* for writing

[2]In this simple example, however, there exists no other instance of class *File* anyway.

Listing 3.1: Specification example: file-io

```
File file;
bool writing;

mock class File{ File(string);
                 bool openWrite();
                 string writeStr(string);
                 bool close()
               }

{
  new(File f)?File(string fname).where(fname != "") {
    file = f;
    !return;
  };
  file?openWrite() {
    writing = true;
    !return(writing);
  };
  while (writing) {
    case {
      file?writeStr(string s) {
        !return(s);
      }
      file?close() {
        writing = false;
        !return(true)
      }
    }
  }
}
```

strings to *file*. A case construct, however, allows the component to alternatively close the file by calling method *close*. This method sets the *writing* status-flag to `false` which causes the specification to leave the while-loop. As a consequence, in particular the component under test must not call *writeStr* anymore.

As mentioned above, the second example, given in Listing 3.2, illustrates a test specification regarding the voter system. Due to the simplicity of the specification language, however, we have to use some additional constructs in the example which are actually not provided by the original specification language. More specifically, we import and use the *Java* classes *HashMap* and *Vector* in order to define the test specification. That is, we assume that the specification language is object-oriented – an extension which is actually discussed in Chapter 5. Recall that the class *Census* is put to test. To this end, a list of *Voter* objects is passed

to an instance of *Census* via method call *conductVoting*. Afterwards we expect the *Census* object to enquire the vote of each of the *Voter* objects by calling their method *vote*. Finally, it should return the conjunction of collected votes.

Similar to the *jMock* specification, we have to create a list of *Voter* objects by means of the standard library class *Vector*. Actually, the specification language does not support the import of library classes. We ignore this problem but initialize the list with three internally created *Vector* objects in Line 6 to 8. Moreover, we define a mapping *votes* which provides the vote for each *Voter* object in terms of Boolean values. For the sake of brevity, we skipped the details of the initialization of the mapping *votes*, but we assume that for each *Voter* object v of voters, the expression $votes.get(v)$ yields a Boolean value which will be used for the object's vote. Furthermore, we create an empty list *called*. During the voting procedure, it will store the object names of the *Voter* object that have been called by the *Census* instance, already.

The main specification statement, starting in Line 17, creates a *Census* object c and calls its method *census* afterwards, passing a copy of the *voters* list to the unit under test. The expectation body of this outgoing method call consists of a while-loop which loops until each *voter* object has been called by c. The body of the while-loop consists of an incoming call expectation of method *vote* of an instance of the *Voter*. Specifically, the where-clause ensures that each *Voter* object is called once, at most. In this case, the object yields its vote consulting the *votes* mapping. Moreover, it calculates the outcome of the voting. Finally, it adds itself to the list *called*.

3.6 Executability and input enabledness

As mentioned earlier, we want to generate an executable test program from a specification. More precisely, for every specification we should be able to automatically derive a *Japl* program which checks whether the unit under test shows the desired behavior at its interface as described by the specification. An important difference between the specification and the resulting test program is that the test program can not enforce the external component to show a certain behavior but instead it tests for it. If the unit shows a behavior that deviates from the specification then the test failed. In particular, we assume that a test program provides some failure handling code which is only executed when the test program detects an unexpected behavior of the unit under test. We don't need to be specific regarding the failure handling code but we only require the code to stop the program from making any progress. Hence, it could merely consist of a diverging while-loop but in real life it would probably report the failure to the user. May it as it be, we refer to the failure handling code by the pseudo statement *fail*. Based on this, we define

Definition 3.6.1 (Test failure detection)**:** Assume $c = (h, \mathsf{v}, (\mu, fail; mc) \circ \mathsf{CS}$ to be a *Japl* configuration whose topmost statement is the pseudo statement *fail*. Then we denote

Listing 3.2: Specification example: voter system

```
1  import java.util.HashMap
2
3  test class Census;
4
5  Census c;
6  Vector voters = new Vector({
7    new Voter(); new Voter(); new Voter();
8  });
9  HashMap votes = ...
10 Vector called = new Vector();
11 Boolean conj = true;
12
13 mock class Voter {
14   Boolean vote();
15 }
16
17 new!Census() {
18   c=?return()
19 };
20 c!conductVote(voters.clone()) {
21   while (called.size() < voters.size()) {
22     (Voter v)?vote().where(called.contains(v) == false) {
23       Boolean myvote = votes.get(v);
24       called.add(v);
25       conj:=conj && myvote;
26       !return(myvote);
27     }
28   }
29   x=?return(Boolean y).where(y == conj)
30 }
```

this with

$$c \downarrow_{\mathsf{fault}} .$$

Moreover, if p is a well-typed *Japl* program with

$$\Delta \vdash p : \Theta \stackrel{s}{\Longrightarrow} \Delta' \vdash c : \Theta' \quad \text{and} \quad c \downarrow_{\mathsf{fault}},$$

then we also may write

$$\Delta \vdash p : \Theta \stackrel{s}{\Longrightarrow} \downarrow_{\mathsf{fault}} .$$

Due to possible test failures, the test program does not have the same trace semantics as the specification. For, as we have seen already, a test program which gave away the control to an external component cannot restrict the incoming communication but is generally input enabled. Therefore, executability means that we can generate a test program which implements the specified outgoing

communication and which, at the same time, detects the first deviation from the specified incoming communication.

Lemma 3.6.2 (Executability)**:** Let s be a specification of our test specification language and let Δ, Θ be an assumption-commitment context such that $\Delta \vdash s : \Theta$. Then there exists a *Japl* program p such that $\Delta \vdash p : \Theta$ and

1. for every trace $t \in [\![\Delta \vdash s : \Theta]\!]$ also $t \in [\![\Delta \vdash p : \Theta]\!]$ as well as
2. (a) for every trace $t\gamma! \in [\![\Delta \vdash p : \Theta]\!]$ also $t\gamma! \in [\![\Delta \vdash s : \Theta]\!]$, and
   (b) for every trace $t\gamma? \in [\![\Delta \vdash p : \Theta]\!]$ either $t\gamma? \in [\![\Delta \vdash s : \Theta]\!]$
   or $\Delta \vdash p : \Theta \stackrel{t\gamma?}{\Longrightarrow}\downarrow_{\mathsf{fault}}$.

Note that within 2.(b) of Lemma 3.6.2, we use an exclusive-or for the two possible cases. That is, the test program p reports a failure if, and only if, the specification did not expect the last incoming communication $\gamma?$. We will prove the executability property in the next chapter by proposing a code generation algorithm which generates a program with the desired properties.

3.7 Satisfiability and completeness

The traces of a test specification's trace semantics describe the behavior that we expect from the unit under test and thus determines what we want to test. But a test which cannot be passed by any program is useless. Therefore, a specification should always describe only traces with incoming communication that is indeed implementable by a program of the programming language. Before we formalize this feature it is important to realize that a change of the viewpoint is involved: in the specification the expected behavior is given in terms of *incoming* communication carried out by an (absent) *external component*. In contrast, saying that a *program* should exist which shows the desired behavior means that the communication shows up in terms of *outgoing* communication within the semantics of the program.

Thus, in order to formalize the satisfiability requirement, we use the dual of a given trace t, denoted by $\bar{t}$, where in each label question marks and exclamation marks are exchanged, such that each incoming communication label becomes an outgoing communication label and vice versa.

Lemma 3.7.1 (Satisfiability)**:** Let s be a specification of our test specification language with $\Delta \vdash s : \Theta$. Then for every trace $t \in [\![\Delta \vdash s : \Theta]\!]$ there exists a *Japl* program p such that $\Theta \vdash p : \Delta$ and $\bar{t} \in [\![\Theta \vdash p : \Delta]\!]$.

Note, that executability requires the existence of a single program, whereas satisfiability involves the existence of a program for each trace. This is a consequence of the input non-determinism introduced by the formal parameters in the incoming communication terms. That is, allowing different incoming values means also allowing different components to pass the test.

The completeness requirement demands that each *possible* behavior of a *Japl* component can be formulated as a *desired* behavior in terms of a specification of the test specification language.

Lemma 3.7.2 (Completeness)**:** Let p be a *Japl* program with $\Delta \vdash p : \Theta$. Then for every trace $t \in [\![\Delta \vdash p : \Theta]\!]$ there exists a specification s such that $\Theta \vdash s : \Delta$ and $\bar{t} \in [\![\Theta \vdash s : \Delta]\!]$.

Chapter 4

Code generation

This chapter describes how to generate a test program of our *Java*-like programming language *Japl*, introduced in Chapter 2, from a test specification given in terms of our test specification language, introduced in Chapter 3. The generation of proper programming language code that implements the specified test is a vital aspect of our testing approach, which is depicted in Figure 4.1. In the left upper corner, the figure sketches a *Japl* component and some environmental *Japl* code which complement one another forming a closed *Japl* program. Component and environment are assumed to communicate, which is represented by the double arrow. Due to the closeness, however, the communication is hidden inside the code. This is indicated by the question mark.

In order to verify, that the component shows the desired behavior to its environment, we first write a specification in terms of our test specification language. The specification represents a simple environment for the component and, at the same time, it phrases the desired behavior by stipulating a required component-environment interaction. This is sketched in the bottom part of the figure, where an exclamation mark within the double arrow indicates that the communication represents a requirement.

As a final step, the test specification is used to generate a *Japl* program which, again, represents an environment for the component and in particular tests for the component's behavior by observing and checking the actual component-environment interaction against the specified behavior.

To understand the general strategy for the generation, it is useful to recapitulate the nature of the specification language and especially, what are the differences to (or additions to) the original programming language. The abstract goal of the specification language is the specification of interaction *traces* used for testing and employing programming-like structuring such as statements, expressions, and method invocations. As far as the *interaction* is concerned, i.e., the calls and returns exchanged at the interface of the unit under test, there is a strong duality between *incoming* and *outgoing* communication, seen from the perspective of the tester. Outgoing calls and returns must be *carried out* by the

Figure 4.1: Testing framework

tester, and incoming communication must be *checked* by it, and both adhering to the *linear order* as given by the specification language, specifying a set of traces. It suggests itself, to realize the interaction labels as given on the *specification* level by corresponding method calls and returns at the *program* level. Obvious as it is, however, to do so requires to tackle the following two points:

control flow: The code at the level of the *Japl* programming language must be contained in bodies of methods, corresponding to the *incoming* method call specifications of the test specification, i.e., the test-code must be appropriately "distributed" over different method bodies and classes. Furthermore and as mentioned, the order of accepting incoming communications and generating outgoing ones must be realized as given by the specification. We use a dynamic labeling mechanism to assure proper interaction *sequencing.*

variable binding: As a consequence of the above mentioned code distribution, we have to deal with the two different *scoping* mechanisms of *method call statements* within the specification language on the one hand, and *method definitions* within *Japl* on the other hand. Although the parameters of an

incoming method call statement at the specification level introduce a scope that resembles the scope of the formal parameters introduced by a method definition at the *Japl* level, there is a crucial difference. For instance, within a specification of two nested incoming call statements[1] the inner call statement may refer to parameters of the outer incoming call statement. At the *Japl* level, however, the two incoming call statements correspond to two method executions which cannot mutually access their formal parameters or variables.

In the following, we will present a code generation algorithm which transforms a test specification of the specification language into *Japl* code. In particular, the algorithm will produce method bodies of tester classes, which implement the specified test. For a better understanding, the algorithm consists of two steps. The first step modifies the specification, in order to introduce the labeling mechanism and to deal with the variable binding problem, respectively. Since the outcome of the transformation is still a specification, it is rather a preprocessing step. The second step, in contrast, will generate method body code from a specification that has been preprocessed already, hence, we can assume certain properties.

4.1 Preprocessing

4.1.1 Labeling mechanism

The programming language *Japl* does not provide language constructs for stating the expectation of a certain incoming communication at a certain point of the program execution. The specification language in contrast provides special expectation statements for this purpose. Recall that the introduction of incoming call statements entails a relaxation of the strict sequential control-flow policy, as these statements are to be processed after realizing an outgoing communication. In *Japl* an outgoing communication always leads to a control context, where the execution of a statement is impossible as the *Japl* program is blocked until an incoming communication occurs. Thus, to stress this specific feature of specification statements that are executed between an outgoing and an incoming communication we introduced the notion of a *passive control context* in Section 3.3 and we, correspondingly, called these statements *passive statements*. Further, recall that apart from incoming call statements we additionally allow while-loops and conditional statements to appear in a passive control context, in order to increase the expressiveness of the specification language.

In particular, the introduction of passive while-loops and conditional statements leads to a *dynamic evaluation* of the incoming communication expectations. That is, the next expected incoming communication is determined at runtime, possibly depending on previous incoming values. This is the basic language

[1] The satisfiability requirement demands an outgoing call statement to occur between the outer and the inner incoming call statement. Though, the outgoing call does not play a role in this example.

disparity that we have to overcome if we want to generate a proper test program in *Japl* that results from a specification of the test specification language.

Our first step on the way to the test program is to introduce the basic framework for ensuring that the external steps carried out by the final test program will occur in the same order as stipulated in the specification. To this end, we tag all incoming communication terms of the specification with a unique identifier. We will use these ids in the final test program in order to match the interface communication steps that occur during the test execution with the corresponding communication statements of the specification. Moreover, the labeling mechanism will enable us to dynamically determine the next expected incoming communication without the need for passive while-loops and conditional statements. This paves the way for generating proper code in the final code generation step, which does not support passive statements.

For a better understanding of the labeling idea, let us take a look at a simple specification snippet:

Listing 4.1: Preprocessing: specification snippet

```
1  u!doSomething(x) {
2    if(e) {
3      (C t)?meth1() { !return(y1); }
4    } else {
5      (C t)t?meth2() { !return(y2); }
6    }
7    ?return(z)
8  };
```

In this example, the method *doSomething* of unit object u is called by the tester and is expected to react with an incoming call of either method *meth1* or method *meth2*, depending on the value of expression e. Both tester methods, *meth1* and *meth2*, immediately return and finally the incoming return from the first method call is expected. For the sake of simplicity, we do not use where-clauses here.

If we want to translate this specification fragment to proper test code of the programming language, we have to face two problems. First, in the operational semantics of the specification language, it is possible to invoke the unit method *doSomething* of u and proceed internally by executing the following conditional statement such that afterwards either the incoming call term of method *meth1* or of *meth2* is on top of the call stack. In the resulting test program, however, reducing the conditional statement right after giving away the control is not possible. Second, in the specification language the incoming call terms express the expectation of either of the methods *meth1* or *meth2*. The programming language, in contrast, does not provide expectation terms but an incoming method call always leads to the execution of the corresponding method body. In particular, basically every method provided by the test program can be called. It is important to understand that, due to this *input-enabledness* of the programming language, we won't be able to generate a program that prevents the tester's environment from showing an undesired behavior. However, the idea is to write a *test* program, that

is, we do not want to prevent the component under test from doing something wrong but we want to *detect* an unexpected behavior. Thus, at least, immediately *after* the call has been accepted, conformance to the specification should be checked, i.e., the invoked method should find out whether it was expected to be called.

Our approach to tackle these problems involves a preprocessing of the specification which is explained in the following by means of the example. First, we annotate the terms for incoming communication with unique ids i_1, i_2, and i_3:

Listing 4.2: Preprocessing: annotated specification

```
1  u!doSomething(x) {
2    if(e) {
3      [i1](C t)?meth1() { !return(y1); }
4    } else {
5      [i2](C t)?meth2() { !return(y2); }
6    }
7    [i3]?return(z)
8  };
```

Furthermore, we introduce a global variable *next* which is used to store the identifier of the next expected incoming communication. Then, in order to determine the next expected call without the passive conditional statement, we have to *anticipate* the conditional statement such that it is implicit decision regarding the next expected call is carried out right *before* the control is given away to the external component. In this example this means we evaluate the conditional expression e and, correspondingly, set the global variable *next* to the identifier of the next expected incoming call term before we call *doSomething*.[2] When the expected method *meth1* or, respectively, *meth2* is invoked then the corresponding expectation body realizes, first, a test on *next* to determine whether this call was expected, i.e., whether it is conform to the specification, and, second, an update of the *next* variable right before the method returns the control back to the tester's environment. In our example both methods have to update *next* to i_3.

As shown below, this leads to an extension of the code by three *next* update statements and three *next* check statement:

Listing 4.3: Preprocessing: anticipation

```
1  if(e) { next = i1 } else { next = i2 };
2  u!doSomething(x) {
3    if(e) {
4      [i1](C t)?meth1() { check(i1); next = i3; !return(y1); }
5    } else {
6      [i2](C t)?meth2() { check(i2); next = i3; !return(y2); }
7    }
8    [i3]?return(z)
```

[2]Note, it is possible to evaluate the expression e earlier, as we assume expressions to be side-effect free.

```
9  };
10 check(i3);
```

In this example three patterns regarding the code generation become apparent:

- Every term which implements an outgoing communication step is immediately preceded by an update of *next*. This applies to outgoing method calls and outgoing returns. The update can be a simple assignment or a rather complex evaluation.

- A passive conditional statement leads to the situation that the preprocessed code contains an equivalent anticipatory conditional statement which implements the update of the *next* variable.

- Every term which implements an incoming communication step is immediately succeeded by a check of *next*. This applies to incoming method calls and incoming returns. We use an auxiliary notation, *check*, for this.

In the final program code, the auxiliary notation *check* will be replaced by a certain statement which implements the test regarding *next*. However, in the specification language it is impossible that the external component implements an unexpected call, anyway. So for the time being we can consider the statement *check* to be equal to ϵ. Yet, we added the check statement in this step already as the check represents the counterpart of the update statement. It also makes the idea of the labeling mechanism more clear. Note, furthermore, that we did not remove the passive conditional statement. The reason is that the preprocessing step shall yield valid specification code. The final program code, naturally, won't contain the passive conditional statement anymore.

Now let us describe a general algorithm for a preprocessing step which transforms test code as sketched in Listing 4.1 into test code as sketched in Listing 4.3. The basic idea is to inspect the passive conditional statements and while-loops of the original code in order to determine a corresponding anticipated update statement of the variable *next*. The resulting code then will consist of the original code, equipped with theses update statements and their corresponding check statements. We define the preprocessing step by a syntax-directed code transformation. The transformation determines all the necessary *next* update statements and, at the same time, inserts these statements, as well as the corresponding checks, into the code. A *next* update statement is an assignment statement of the following form:

$$s_{nxt} ::= next = e \mid \texttt{if}(e)\ \{s_{nxt}\}\ \texttt{else}\ \{s_{nxt}\}$$

Remark 4.1.1: Within a specification that provides the global variable *next*, the execution of a next update statement s_{nxt} always terminates. Specifically, apart from the assignment to *next*, it is free of side-effects.

Since the specification language allows nestings of passive conditional and while statements, a *next* update statement might equally consist of nested conditional statements. During the preprocessing's recursive descent through the specification an update statement might evolve until it is finally inserted at its intended position in the code. We define two mutually recursively applied functions

$$\begin{aligned} prep_{in} &: s^{psv} \times s_{nxt} \rightarrow s_{nxt} \times s^{psv} \quad \text{and} \\ prep_{out} &: s^{act} \rightarrow s^{act}, \end{aligned}$$

given in Table 4.1 and Table 4.2, respectively. Both functions expect a statement as argument which is in passive or, respectively, active control context. They return the same statement but annotated with ids as well as extended by checks and *next* update statements. Additionally, as a second argument, $prep_{in}$ expects a *next* update statement which determines the identifier of the next incoming communication that is expected to happen *after* statement s^{psv} has been executed. The update statement is inserted in s^{psv} in front of its last outgoing return. Dually, $prep_{in}$ also yields a new *next* update statement which describes the next expected incoming call of s^{psv} itself, i.e., which has to be carried out *before* s^{psv} is executed.

$$\begin{aligned}
&prep_{out}(e!m(e,\ldots,e)\{\overline{T}\,\overline{x};\ s_1^{psv};\ x = ?\texttt{return}(T\,x').\texttt{where}(e')\ \}) \stackrel{\text{def}}{=} \\
&\quad s_{nxt};\ e!m(e,\ldots,e)\{\overline{T}\,\overline{x};\ s_2^{psv};\ x = [i]?\texttt{return}(T\,x').\texttt{where}(e')\};\ check(i,e'); \\
&\qquad \text{where}\ \ (s_{nxt}, s_2^{psv}) = prep_{in}(s_1^{psv}, \mathsf{next} = i) \\
\\
&prep_{out}(\texttt{new}!C(e,\ldots,e)\{\overline{T}\,\overline{x};\ s_1^{psv};\ x = ?\texttt{return}(C\,x').\texttt{where}(e')\ \}) \stackrel{\text{def}}{=} \\
&\quad s_{nxt};\ \texttt{new}!C(e,\ldots,e)\{\overline{T}\,\overline{x};\ s_2^{psv};\ x = [i]?\texttt{return}(C\,x').\texttt{where}(e')\};\ check(i,e'); \\
&\qquad \text{where}\ \ (s_{nxt}, s_2^{psv}) = prep_{in}(s_1^{psv}, \mathsf{next} = i) \\
\\
&prep_{out}(\texttt{if}\ (e)\ \{s_1^{act}\}\ \texttt{else}\ \{s_2^{act}\}) \stackrel{\text{def}}{=} \\
&\quad \texttt{if}\ (e)\ \{prep_{out}(s_1^{act})\}\ \texttt{else}\ \{prep_{out}(s_2^{act})\} \\
\\
&prep_{out}(\texttt{while}\ (e)\ \{s^{act}\}) \stackrel{\text{def}}{=} \texttt{while}\ (e)\ \{prep_{out}(s^{act})\} \\
\\
&prep_{out}(s_1^{act};\ s_2^{act}) \stackrel{\text{def}}{=} prep_{out}(s_1^{act});\ prep_{out}(s_2^{act}) \\
\\
&prep_{out}(\{\overline{T}\,\overline{x};\ s^{act}\}) \stackrel{\text{def}}{=} \{\overline{T}\,\overline{x};\ prep_{out}(s^{act})\} \\
\\
&prep_{out}(x = e) \stackrel{\text{def}}{=} x = e
\end{aligned}$$

Table 4.1: Preprocessing: labeling and anticipation ($prep_{out}$)

The definition of $prep_{out}$ is straightforward. Its solely interesting case deals with an outgoing call statement. The call's incoming return term is annotated

with a new identifier i[3]. The return value of $prep_{out}$ comprises not only a modified version of the call statement but it represents actually a sequence of three statement: the call statement is framed by an anticipating *next* update statement and a check statement. In order to find out the proper update statement, however, the function $prep_{in}$ must be applied on the body of the call expectation statement. The application of $prep_{in}$ also inserts the return term's update statement into the expectation body, which is merely an assignment of i to *next*. For all other active statements, $prep_{out}$ is either the identity or, in case of a composite statement, $prep_{out}$ is applied recursively.

$$prep_{in}((C\ x)?m(\overline{T}\ \overline{x}).\texttt{where}(e)\{\overline{T}\ \overline{x};\ s^{act};\ \texttt{!return}\ e'\}, s_{nxt}) \stackrel{\text{def}}{=} (next{=}i,\ [i]\,(C\ x)?m(\overline{T}\ \overline{x}).\texttt{where}(e)\{\overline{T}\ \overline{x};\ check(i,e);\ prep_{out}(s^{act});\ s_{nxt}\ \texttt{!return}\ e'\})$$

$$prep_{in}(\texttt{new}(C\ x)?C(\overline{T}\ \overline{x}).\texttt{where}(e)\{\overline{T}\ \overline{x};\ s^{act};\ \texttt{!return}\}, s_{nxt}) \stackrel{\text{def}}{=} (next{=}i,\ [i]\,(C\ x)?m(\overline{T}\ \overline{x}).\texttt{where}(e)\{\overline{T}\ \overline{x};\ check(i,e);\ prep_{out}(s^{act});\ s_{nxt}\ \texttt{!return}\})$$

$$prep_{in}(\varepsilon, s_{nxt}) \stackrel{\text{def}}{=} (s_{nxt}, \varepsilon)$$

$$prep_{in}(\texttt{if}(e)\{s_1^{psv}\}\ \texttt{else}\ \{s_2^{psv}\}, s_{nxt}) \stackrel{\text{def}}{=} (\texttt{if}(e)\{s_{nxt}^1\}\ \texttt{else}\ \{s_{nxt}^2\}, \texttt{if}(e)\ \{\tilde{s}_1^p\}\ \texttt{else}\ \{\tilde{s}_2^p\})$$

where

$$(s_{nxt}^1, \tilde{s}_1^p) = prep_{in}(s_1^{psv}, s_{nxt}) \quad \text{and} \quad (s_{nxt}^2, \tilde{s}_2^p) = prep_{in}(s_2^{psv}, s_{nxt})$$

$$prep_{in}(s_1^{psv}; s_2^{psv},\ s_{nxt}) \stackrel{\text{def}}{=} (\ s_{nxt}^1,\ \tilde{s}_1^p; \tilde{s}_2^p\ )$$

where

$$(s_{nxt}^2, \tilde{s}_2^p) = prep_{in}(s_2^{psv}, s_{nxt}) \quad \text{and} \quad (s_{nxt}^1, \tilde{s}_1^p) = prep_{in}(s_1^{psv}, s_{nxt}^2)$$

$$prep_{in}(\texttt{while}(e)\{s^{psv}\}, s_{nxt}) \stackrel{\text{def}}{=} (\ \texttt{if}(e)\ \{s_{nxt}^1\}\ \texttt{else}\ \{s_{nxt}\},\ \texttt{while}(e)\ \{\tilde{s}^p\}\ )$$

where

$$(s_{nxt}^1, _) = prep_{in}(s^{psv}, s_{nxt}) \quad \text{and} \quad (s_{nxt}^2, \tilde{s}^p) = prep_{in}(s^{psv}, \texttt{if}(e)\{s_{nxt}^1\}\ \texttt{else}\ \{s_{nxt}\})$$

$$prep_{in}(\texttt{case}\ \{\overline{stmt_{in}; s^{psv}}\}, s_{nxt}) \stackrel{\text{def}}{=} (\ next = i,\ \texttt{case}\ \{\overline{\tilde{stmt}_{in}; \tilde{s}^p}\}\ )$$

where for each $stmt_{in}^l; s_l^{psv} \in \overline{stmt_{in}; s^{psv}}$

$$stmt_{in}^l = (C\ x)?m(\overline{T}\ \overline{x}).\texttt{where}(e)\{\overline{T}\ \overline{x}; s^{act};\ \texttt{!return}\ e'\} \quad \text{it is}$$

$$(s_{nxt}^l, \tilde{s}_l^p) = prep_{in}(s_l^{psv}, s_{nxt}) \quad \text{and}$$

$$\tilde{stmt}_{in}^l = [i](C\ x)?m(\overline{T}\ \overline{x}).\texttt{where}(e)\{\overline{T}\ \overline{x}; check(i,e);\ prep_{out}(s^{act});\ s_{nxt}^l\ \texttt{!return}\ e'\}$$

Table 4.2: Preprocessing: labeling and anticipation ($prep_{in}$)

As shown in Table 4.2 the function $prep_{in}$, applied to an incoming method or constructor call, annotates the call with a new identifier, puts the given update statement s_{nxt} in front of the outgoing return, and yields an assignment to i as its own update statement. Moreover, it applies $prep_{out}$ to the expectation body.

[3]We assume a unique name generation scheme here which guarantees that the new identifier is indeed not used within the rest of the program.

As we have seen in the example, regarding the update statement to be calculated, a passive conditional statement leads to a conditional update statement. Note that $prep_{in}$ is applied recursively to the conditional's branches which yield to corresponding *next* update statements that have to be incorporated into the conditional update statement. Moreover, the given update statement that is to be inserted, has to be inserted in both branches of the passive conditional statement.

Regarding sequential composition, the given update statement s_{nxt} has to be inserted in s_2^{psv} since s_{nxt} determines the next incoming communication that happens after the sequential composition. The processing of s_2^{psv} yields a new update statement that has to be inserted in s_1^{psv} whose transformation, in turn, yields the final update statement for the whole sequence.

Processing of the while-loop leads to two recursive applications of $prep_{in}$. The first call is used to find out the update statement solely for the while-body. In particular, we are not in interested in the resulting code transformation. This is indicated by the _ symbol. However, if the expression e is false then the body of the while-loop would be skipped. Thus the update statement of the while-loop is a conditional statement, where one branch consists of the update statement of the while-loop body and the other one of the update statement of the consecutive statement. The resulting update statement has to be inserted also in the body statement itself which is done by the second application of $prep_{in}$.

The processing of the case statement, finally, follows the pattern of the processing of an incoming call. Nevertheless, we do not apply $prep_{in}$ recursively, as we want to equip every call of the case statement with the same expectation identifier. This way, we express that each branch of the case statement represents an expected interface communication.

Note that the transformation functions are well-defined. More specifically, $prep_{in}$ is defined for all statements that may occur in a passive context and $prep_{out}$ for all statements that may occur in an active control context. The mutual recursion regarding the body of call statements is justified by Remark 3.3.2. Moreover, it is easy to see that the resulting code is syntactically correct and well-typed (under the assumption that the original statement was syntactically correct and well-typed and that the resulting program is extended by the global variable *next*).

A specification that results from the preprocessing step mentioned above has the following properties:

- Each incoming method call statement is of the following form:

  $$[i]\,(C\,x)?m(\overline{T}\,\overline{x}).\texttt{where}(e)\{\overline{T}\,\overline{x};\ check(i,e);\ s^{act};\ s_{nxt}\ !\texttt{return}\ e'\},$$

  that is, the call is annotated with an identifier, the body starts with a corresponding expectation check, and the return term is preceded by an expectation update statement. The identifier is unique unless the call is a branch of a case expression, where other calls with the same identifier annotation could exist.

  The incoming constructor call has the same features.

- Each outgoing call statement is transformed into code of the following form:

  $$s_{nxt};\ e!m(e,\ldots,e)\ \{\overline{T}\,\overline{x};\ s^{psv};\ [i]\,x = ?\texttt{return}(T\,x').\texttt{where}(e')\};\ check(i,e'),$$

  where i is a unique identifier. Moreover, within the specification, each occurrence of an outgoing call statement is preceded by an update statement s_{nxt} and followed by an expectation check $check(i,e')$.

  The same properties hold for an outgoing constructor call.

Remark 4.1.2 (Adjustment of initial expectation identifier)**:** Consider, we want to preprocess a specification

$$s = \overline{cutdecl}\ \overline{T}\ \overline{x};\ \overline{mokdecl}\ \{\ stmt\ \},$$

where the body statement $stmt$ is passive. Applying $prep_{in}$ to $stmt$ yields not only the new statement, $stmt'$, but also a $next$ update statement s_{nxt}. In order to anticipate the next incoming communication of $stmt'$, the update statement s_{nxt} has to be executed at the very beginning of the specification. Since the specification body appears in a passive control context, however, this is not possible.

The solution is as follows. Assume T to be the type of the expectation identifiers as well as of the variable $next$ and $i_0 = ival(T)$. That is, the operational semantics initializes each variable of type T to i_0. Within the preprocessed specification, we replace all occurrences of identifier i by the initial value i_0 where i is determined by the execution of s_{nxt}:

$$c_{init}(\overline{T}\,\overline{x};\ T\ next;\ \{s_{nxt};\ \texttt{return}\}) \longrightarrow^* (h, \mathsf{v}, (\mu, \texttt{return})) \quad \text{and}$$
$$i = [\![next]\!]_h^{\mathsf{v},\mu}.$$

Renaming the identifier of the first expected incoming communication to the initial value i_0 leads to the fact that we do not need to explicitly initialize $next$ with a specific value.

Note, that s_{nxt} always consists of conditional statements and assignments to $next$, only. In particular, it does not involve any loops or method or constructor calls. Thus, the small program above that executes s_{nxt} for determining the very first expectation identifier always reaches the terminal configuration.

The main idea of the preprocessing is the following. Whenever an incoming communication expectation term is about to be executed, its associated identifier is indeed stored in the global variable $next$. In other words, whenever an incoming call or return occurs the variable $next$ indicates whether this call or return was expected. This is formalized by the following lemma.

Lemma 4.1.3 (Anticipation)**:** Let s be a valid specification and $stmt$ its body statement. Then let s' be the specification that results from the preprocessing step such that we introduce a new global variable $next$ and the body statement of s is replaced by either $prep_{out}(stmt)$ or $prep_{in}(stmt)$, depending on whether $stmt$ is an active or a passive

statement. (If *stmt* is passive additionally consider an adjustment of the initial expectation identifier according to Remark 4.1.2). Let, in particular, $c = (h, \mathsf{v}, (\mu, mc) \circ \mathsf{CS})$ be a configuration such that

$$\Delta \vdash c_{init}(s') : \Theta \stackrel{t}{\Longrightarrow} \Delta' \vdash c : \Theta' \,.$$

Then the following holds:

1. if $mc = [i]\,(C\,x)?m(\overline{T}\,\overline{x})\{\ldots\};\ mc^{act}$ then $[\![next]\!]_h^{\mathsf{v},\mu} = i$,
2. if $mc = [i]\,\mathtt{new}(C\,x)?(\overline{T}\,\overline{x})\{\ldots\}; mc^{act}$ then $[\![next]\!]_h^{\mathsf{v},\mu} = i$
3. if $mc = \mathtt{case}\ \{[i]\,\overline{stmt}\ \};\ mc^{act}$ then $[\![next]\!]_h^{\mathsf{v},\mu} = i$, and
4. if $mc = [i]?\mathtt{return}(T\,x).\mathtt{where}(e);\ mc^{act}$ then $[\![next]\!]_h^{\mathsf{v},\mu} = i$.

Remember, however, that the variable *next* does not have any influence on the behavior of the preprocessed specification. For, no statement but only the new *check* statements evaluates *next* in order to test if the actual incoming communication matches with the specification. Since the preprocessed specification still contains the original expectation statement, which do not accept a wrong behavior anyway, these checks are always positive. As mentioned early, we will need *next* in the final *Japl* program due to the general input-enabledness of the programming language.

4.1.2 Variable binding

The specification language supports nested incoming and outgoing call statements such that formal parameters and local variables of outer statements are accessible also within the inner statements. This supports the look-and-feel of the original programming language where also static scopes for local variable declaration exist. Since we have to move and distribute most of the specification code into method bodies, however, the original local scopes do not exist in the resulting code anymore, rendering it impossible to access certain local variables or formal parameters. Listing 4.4 shows a small specification snippet which has been already preprocessed regarding the expectation identifiers, i.e., the code is already annotated with identifiers. The example shows two nested incoming call statements. For the sake of simplicity, both calls address the same class and method. The first incoming call defines a formal parameter x_p as well as a local variable x_l and the second one only a parameter y_p. Thus, the body of the second call statement has access to both the parameters x_p and y_p as well as to the local variable x_l. The inner call statement, indeed, makes usage of the outer call's local variable x_l within its where-clause and also it accesses the outer call's formal parameter x_p within a conditional statement.

In order to get valid test code, we have to translate the two incoming call statements into code which will reside in the method body of method *meth*. In particular, the translation of the sketched conditional statement will be part of

Listing 4.4: Formal parameters and local variables

```
[i] (C x)?meth(C x_p) {
  C x_l;
  ...
  [j] (C y)?meth(C y_p).where(y_p > x_l) {
    if(x_p < y_p) { ... }
    ...
  }
}
```

the method's body. However, the second invocation of *meth* won't be aware of the variable x_p. In order to make it accessible, we have to make the variable *globally* accessible. To this end we extend our preprocessing step with the introduction of global variables x_p^g, x_l^g, and y_p^g representing the counterpart of the local variables x_p, x_l, and y_p, respectively. Additionally, we introduce global counterparts x^g and y^g for the callees of the two incoming calls. Right after the first invocation of *meth*, the expectation body has to assign the values of its actual parameters to x^g and x_p^g. When the method is called a second time, the global variable x_p^g is used to access the value of the formal parameter of the first call. Furthermore the global variable x_l^g is used in the where-clause. The result is shown in Listing 4.5. Note that we still use the "local" parameter y_p in the where-clause as its value has not been copied to y_p^g when the clause is evaluated.

Listing 4.5: Variable globalization

```
[i] (C x)?meth(C x_p) {
  x^g=x; x_p^g=x_p;
  ...
  [j] (C y)?meth(C y_p).where(y_p > x_l^g) {
    y^g=y; ...
    if(x_p^g < y_p^g) { ... }
    ...
  }
}
```

Since the general "variable globalization step", as it has been explained by the example above, is rather straightforward, we don't want to introduce it in all its formal details but we sketch the basic idea. In general we extend our preprocessing of specification programs by the following steps:

- For each local variable and formal parameter that occur in the original specification, a new global variable is added.[4]

[4]We assume all local variables and formal parameters of the original specification to be

- Each incoming call or return term is followed by a sequence of statements that copy the values of the formal parameters to their global correspondent. In the following we will refer to this sequence by the auxiliary statement s_{vinit} if needed.
- Each occurrence of a local variable or parameter within the specification is replaced by its global correspondent. This, of course, neither applies to the occurrences of formal parameters in the incoming call or return term itself nor to the occurrences in s_{vinit}.
- A consequence is that local variable are of no use anymore, hence, we remove all local variable declarations within expectation statements and block statements. Specifically, block statements $\{\ \overline{T}\,\overline{x};\ \mathit{stmt}\ \}$ are resolved in that they are replaced by their wrapped statement *stmt*.

Having explained separately the two main aspects of the preprocessing step we bundle them by means of a definition.

Definition 4.1.4 (Preprocessing)**:** Consider

$$s = \overline{\mathit{cutdecl}}\ \overline{T}\ \overline{x};\ \overline{\mathit{mokdecl}}\ \{\mathit{stmt}\},$$

to be a valid specification. Then with $\mathit{prep}(s)$ we denote the specification

$$s' = \overline{\mathit{cutdecl}}\ \overline{T}\ \overline{x};\ \overline{T'}\ \overline{x'};\ \overline{\mathit{mokdecl}}\ \{\mathit{stmt}'\},$$

that results from preprocessing s. In particular, depending on the control context, the body statement stmt' results from either applying prep_{out} or prep_{in} to stmt followed by a variable globalization as explained above. Hence, the new variables $\overline{x'}$ comprise *next* as well as the global counterparts of the formal parameters and local variables defined in *stmt*. In case that *stmt* is passive we additionally consider an adjustment of the initial expectation identifiers as explained in Remark 4.1.2.

4.2 *Japl* code generation

We have seen that the preprocessing step results in a specification which contains a global variable *next* that is updated to the identifiers of the next expected *incoming* communication — right *before* the specification passes the control to the component through an *outgoing* communication. Moreover, due to variable globalization the specification is free from variable accesses crossing an outgoing communication. These were important steps towards the final test program. However, the preprocessed specification still contains expectation statements, which do not exist in the programming language *Japl*. In the next step we finally translate these statements to syntactic valid *Japl* code.

Before we start, let us summarize the features of a specification which results from the preprocessing step that was described above.

different. Otherwise we can accomplish this by a proper renaming as we consider *Var* to be infinite.

1. The list of the specification's global variables includes the variable *next* and global correspondents for all formal parameters and local variables of the original specification.

2. All accesses to local variables and formal parameters within the original specification are "redirected" to the corresponding global variable, i.e., all occurrences of local variables and formal parameters within assignments and expressions of the original specification are replaced by their global counterparts.

3. The specification is free from local variables and free from block statements.

4. An incoming method call statement always has the following form:

$$[i]\,(C\,x)?m(\overline{T}\,\overline{x}).\texttt{where}(e)\{s_{vinit};\ check(i,e);\ s^{act};\ s_{nxt};\ !\texttt{return}\ e'\}.$$

   That is, the body consists of a statement s_{vinit} that assigns the values of the actual parameters to global variables, a check whether this call was expected, the actual body s^{act}, an expectation update statement s_{nxt}, and finally the return term. In particular, the body does not introduce any local variables. Incoming constructor call statements and case statements have a similar form.

5. Each outgoing method call statement always appears in following form:

$$s_{nxt};\ e!m(e,\ldots,e)\{s^{act};\ [i]\,x=?\texttt{return}(x').\texttt{where}(e')\};\ check(i,e'),$$

   such that each call statement is preceded by an expectation update statement and followed by a check. Outgoing call statements do not introduce local variables either.

As mentioned before, the last thing that remains to be done is to remove the passive statements and to translate the expectation statements into valid code of the programming language. As for the incoming call statements, the basic idea is to move the expectation body into the method body of the callee method. However, in order to do so, we have to consider the following:

- If the specification contains two or more incoming call statements that address the same method, then we have to add all the corresponding expectation bodies to the same method body. Thus, we have to make sure that the corresponding *Japl* code of either of the expectation bodies is executed each time the method is called. In particular, exactly the expectation body must be chosen that matches with the specification at the specific situation where the call occurs. Moreover, if the method is called but no matching expectation statement of the specification can be found, the test program should realize this and consider it to be an unexpected behavior.

Listing 4.6: Code generation: method body scheme

T *meth*($T_1\ x_1, \ldots, T_n\ x_n$) {
  T *retVal*;

expectation$_1$
$\vdots$
expectation$_k$
fail;

  `return`(*retVal*);
}

- In the specification, each incoming call statement introduces its own set of formal parameters. A method definition, however, provides only one set of formal parameters. Since more than one incoming call statements might flow into a single method definition, the call statement's formal parameters have to be unified.

- Certainly, we cannot merely copy an expectation body into the corresponding method body, as in general an expectation body might contain a nesting of other expectation statements, which have to be translated as well. Moreover, the programming language does allow exactly only one return term at the end of a method definition. Thus we cannot add a return term for each expectation body.

Listing 4.6 sketches our approach for the generation of method code. A method body always starts with the definition of a local variable *retVal* which is used for the return value. For each of the method's call expectation statements we put the corresponding method code, represented by the expectation$_i$ boxes, between the variable definition and the return statement. More precisely, the expectation boxes are actually nested and this nesting ends with the pseudo statement *fail* which represents the error handling in case of an unexpected call. Listing 4.7 sketches the *Japl* code that implements an incoming call statement, that is, it shows how the expectation boxes of Listing 4.6 look like. The nesting arises from the fact that each expectation handler is wrapped into a conditional statement which checks whether the actual call of the method matches with the incoming call expectation statement. Thus the corresponding code is executed only if the variable *next* holds the identifier of the incoming call statement *and* if the expression of the where-clause evaluates to true. In this case the actual code of the expectation body is executed and finally the return variable *retVal* is set to the return value of the call statement. Otherwise, we have to check the other expectation handlers. If even the inner-most expectation does not match with the actual call, then the call was unexpected, that is, the else-branch of the inner-most expectation box consists of

Listing 4.7: Code generation: code for $\boxed{\text{expectation}_{k-1}}$

```
1 if((next == [ id ]) && [ check-where-clause ]) {
2   [ body ]
3   retVal = [ ret-val ];
4 } else {[ expectation_k ]};
```

the *fail* statement.

The constructor of a class has a similar pattern. As the return value of a constructor is always the new instantiated object, however, we do not have to provide a return variable in the constructor body. In exchange we have to deal with internal object creation. Thus, constructor bodies differ from method bodies in that they additionally contain a conditional statement which enables internal calls:

```
if(internal == true) {
  skip;
} else { fail };
```

Thus, if no matching incoming call expectation can be found, then, before we consider the constructor call to be unexpected, we additional check if an *internal* object creation was expected. To this end, we consult a dedicated global Boolean variable *internal*. A value of `true` indicates an internal constructor call that corresponds to an equivalent call within the specification. In this case the constructor has to do nothing but solely return the new object since the specification language does not allow to provide specific code for internal object creation. Accordingly, every internal object creation, $x = \texttt{new}\ C$, within the specification program will be translated to a similar object creation framed by assignments to the new global variable *internal*:

```
internal = true;
x = new C(v_1, ... v_k);
internal = false;
```

This way, the constructor can distinguish internal calls from unexpected incoming calls. Note that due to typing issues it might be necessary to provide some dummy parameter values v_1 to v_k. As shown above, the internal object creation always results in the execution of the empty statement *skip* only, such that actual values of the dummy parameter have no influence on the new object.

In the following, we assume a set of class definitions $\overline{cldef}$ which consists of the classes to be provided by the tester program. Each of the classes' methods is of the structure as shown in Listing 4.6. We will present an iterative transformation algorithm which will extend the method bodies piece by piece but we will start with classes where each method and constructor body does not contain any expectation code so far. That is, we assume a set of initial class definitions

method code:	constructor code:
$T\ meth(T_1\ x_1, \ldots, T_k\ x_k)\ \{$	$C(T_1\ x_1, \ldots, T_k\ x_k)\ \{$
$\quad T\ retVal;$	
	$\quad \texttt{if}(internal == \texttt{true})\ \{$
	$\quad\quad skip;$
$\quad fail;$	$\quad \}\ \texttt{else}\ \{\ fail;\ \}$
$\quad \texttt{return}(retVal);$	$\quad \texttt{return};$
$\}$	$\}$

Table 4.3: Initial method and constructor code

$\overline{cldef}_{init}$ where each method and constructor of the classes is of the form as shown in Table 4.3.

As mentioned above, starting from these initial class definitions we gradually extend the method and constructor bodies in order to add code that deals with a certain call expectation. Table 4.4 introduces an auxiliary notation which describes the modification of a class definition set by extending a method body with call expectation code. The notation

$$\overline{cldef}.C.m \overset{(i,e_w)}{\rightarrow} stmt : e$$

represents a sequence of class definitions which is identical to $\overline{cldef}$ except that the method body of method m of class C is extended by the statement $stmt$. More precisely, a new conditional statement as sketched in Listing 4.7 is created where i represents the expected communication identifier and e_w is the expression of the where-clause. Along with a return value assignment, the new statement $stmt$ is inserted as the main branch of the conditional statement whereas the original

$$\begin{array}{l}
\overline{cldef}.C.m \overset{(i,e_w)}{\rightarrow} stmt : e \overset{\text{def}}{=} \overline{cldef'} \quad \text{where} \\
\quad cldef = \texttt{class}\ C\ \{\overline{T}\ \overline{f}\ \overline{mdef}\} \in \overline{cldef}, \\
\quad mdef = T'\ m(\overline{T'}\ \overline{x'})\{\overline{T_l}\ \overline{x_l};\ stmt_b;\ \texttt{return}(retVal)\} \in \overline{mdef}, \\
\quad mdef' = T'\ m(\overline{T'}\ \overline{x'})\{\ \overline{T_l}\ \overline{x_l}; \\
\qquad\qquad\qquad \texttt{if}((next == i)\ \&\&\ (e_w))\{ \\
\qquad\qquad\qquad\quad stmt;\ retVal = e; \\
\qquad\qquad\qquad \}\ \texttt{else}\ \{\ stmt_b;\ \}; \\
\qquad\qquad\qquad \texttt{return}(retVal)\ \}, \\
\quad cldef' = \texttt{class}\ C\ \{\overline{C}\ \overline{x}\ \overline{mdef'}\} \in \overline{cldef}, \\
\quad \overline{cldef'} = \overline{cldef} \setminus \{cldef\} \cup \{cldef'\}.
\end{array}$$

Table 4.4: Code-generation: method extension

method body statement forms the else-branch. That is, in the definition we exploit our knowledge about the structure of the method bodies. Note, that the meaning of the notation is not defined for sequences of class definitions where class C does not exist or where class C does not provide an appropriate method definition. We use $\overline{cldef}.C.C \overset{(i,e_w)}{\rightarrow} stmt$ for the extension of a constructor body which only differs from the definition given in Table 4.4, in that we do not add a return value assignment.

The final code generation step is carried out by two mutual recursive functions, which are pointwise defined in terms of simple functional programming code for the sake of clarity. The function

$$code_{out} : \overline{cldef} \times s_{out} \rightharpoonup \overline{cldef} \times stmt^{pl},$$

given in Table 4.5, generates code only from specification statements which are in active control context. It yields a statement of the programming language equivalent to the original specification statement.[5] However, the function additionally returns a new class definition sequence. For, the specification statement could incorporate an expectation statement resulting in the extension of the corresponding callee class. The function

$$code_{in} : \overline{cldef} \times s_{in} \rightharpoonup \overline{cldef}$$

transforms statements that are in passive control context into method body code, modifying the given set of class definitions. The function's definition is given in Table 4.6.

Let us have a closer look at the $code_{out}$ definition. The first two definitions of Table 4.5 deal with outgoing method and constructor call statements. When we translate such a call statement of the specification language into proper programming language code, we have to merge the expectation statement's call term with its return term to get a call statement of the programming language. Moreover, the specification body must be processed. As the specification body might contain incoming call expectations, its processing potentially leads to a modification of the given class definitions. Note, that we assume a specification which has been preprocessed, that is, we do not need to add a check regarding the expectation identifier or regarding the where-clause, since the preprocessing has already added it. Moreover, we can assume that the specification code does not contain declarations of local variables.

The transformation does not need to modify assignments. As explained above, internal object creations have to be distinguishable from unexpected incoming constructor calls. Thus, the translation uses a corresponding flag to indicate an internal instantiation. Moreover, we have to add dummy parameters to the constructor call, in order to get a well-typed call. For each parameter of type T we

[5] We use the superscript pl to indicate that the resulting statement is an element of the programming language.

$$
\begin{array}{l}
code_{out}(\ \overline{cldef},\ e!m(\overline{e})\ \{stmt;\ [i]?\texttt{return}(x).\texttt{where}(e_w)\}\ ) \stackrel{\text{def}}{=} \\
\quad \mathsf{let}\ \ \overline{cldef'} = code_{in}(\overline{cldef}, stmt)\ \ \mathsf{in}\ (\ \overline{cldef'},\ x{=}e.m(\overline{e})\ ). \\
\\
code_{out}(\ \overline{cldef},\ \texttt{new}!C(\overline{e}\ ))\{stmt;\ [i]?\texttt{return}(x).\texttt{where}(e_w)\}) \stackrel{\text{def}}{=} \\
\quad \mathsf{let}\ \ \overline{cldef'} = code_{in}(\overline{cldef}, stmt)\ \ \mathsf{in}\ (\ \overline{cldef'},\ x{=}\texttt{new}\ C(\overline{e})\ ). \\
\\
code_{out}(\ \overline{cldef},\ x{=}e\ ) \stackrel{\text{def}}{=} (\ \overline{cldef},\ x{=}e\ ) \\
\\
code_{out}(\ \overline{cldef},\ \texttt{new}\ C()\ ) \stackrel{\text{def}}{=} \\
\quad \mathsf{let}\ \ \overline{T}\ \overline{x} = cparams(C)\ \ \mathsf{in} \\
\quad\quad \mathsf{let}\ \ stmt = intern{=}\texttt{true};\ x = \texttt{new}\ C(ival(\overline{T}));\ intern{=}\texttt{false}\ \mathsf{in}\ (\ \overline{cldef},\ stmt\ ). \\
\\
code_{out}(\ \overline{cldef},\ stmt_1;\ stmt_2\ ) \stackrel{\text{def}}{=} \\
\quad \mathsf{let}\ (\overline{cldef_1}, stmt_1^{pl}) = code_{out}(\overline{cldef}, stmt_1)\ \ \mathsf{in} \\
\quad\quad \mathsf{let}\ (\overline{cldef_2}, stmt_2^{pl}) = code_{out}(\overline{cldef_1}, stmt_2)\ \ \mathsf{in}\ (\ \overline{cldef_2},\ stmt_1^{pl};\ stmt_2^{pl}\ ). \\
\\
code_{out}(\ \overline{cldef},\ \texttt{while}\ (e)\ \{stmt\}\ ) \stackrel{\text{def}}{=} \\
\quad \mathsf{let}\ (\overline{cldef'}, stmt^{pl}) = code_{out}(\overline{cldef}, stmt)\ \ \mathsf{in}\ (\ \overline{cldef'},\ \texttt{while}\ (e)\ \{stmt^{pl}\}\ ). \\
\\
code_{out}(\ \overline{cldef},\ \texttt{if}\ (e)\ \{stmt_1\}\,\texttt{else}\ \{stmt_2\}\ ) \stackrel{\text{def}}{=} \\
\quad \mathsf{let}\ (\overline{cldef_1}, stmt_1^{pl}) = code_{out}(\overline{cldef}, stmt_1)\ \ \mathsf{in} \\
\quad\quad \mathsf{let}\ (\overline{cldef_2}, stmt_2^{pl}) = code_{out}(\overline{cldef_1}, stmt_2)\ \ \mathsf{in}\ (\overline{cldef_2},\ \texttt{if}(e)\ \{stmt_1^{pl}\}\,\texttt{else}\{stmt_2^{pl}\})
\end{array}
$$

Table 4.5: Generation of *Japl* code ($code_{out}$)

pass its initial value $ival(T)$ to the constructor. The parameter types can be looked up in the class definition of the corresponding class.

A sequence of two active expectation statements is processed by transforming each statement, i.e. a sequence is processed in terms of two recursive applications of $code_{out}$. We pass the original class definitions to the $code_{out}$ application regarding the first statement and we use the resulting class definition for the transformation of the second statement. The class definitions that result from the second transformation then represents also the result of the sequence' transformation. While-loops, and conditional statements are processed similarly, that is, we have to process their sub-statements, recursively.

Now let us discuss the definitions of $code_{in}$ of Table 4.6. Again the processing of incoming method and incoming constructor calls are similar. One common task is to substitute the expectation statement's formal parameters by the formal parameters of the corresponding method or constructor definition and the expectation's callee names by the special self reference symbol `this`, respectively.

The remaining passive statements are compositions of other passive statements, hence, the transformation is realized by recursive applications of $code_{in}$.

$$
\begin{array}{l}
\mathit{code}_{in}(\overline{\mathit{cldef}}, [i]\,(C\;x)?m(\overline{T}\;\overline{x}).\texttt{where}(e)\{\mathit{check}(i,e);\;\mathit{stmt};\;!\texttt{return}\;e_r\}) \stackrel{\text{def}}{=} \\
\quad \mathsf{let}\;(\overline{\mathit{cldef}'}, \mathit{stmt}^{pl}) = \mathit{code}_{out}(\overline{\mathit{cldef}}, \mathit{stmt})\;\;\mathsf{in} \\
\qquad \mathsf{let}\;\;\overline{T}\;\overline{x}_p = \mathit{mparams}(C,m)\;\;\mathsf{in} \\
\qquad\quad \mathsf{let}\;(e', e'_r, \mathit{stmt}^{pl\prime}) = (e, e_r, \mathit{stmt}^{pl})[\texttt{this}/x,\;\overline{x}_p/\overline{x}]\;\;\mathsf{in}\;\;\overline{\mathit{cldef}'}.C.m \stackrel{(i,e)}{\rightarrow} \mathit{stmt}^{pl\prime} : e'_r. \\
\\
\mathit{code}_{in}(\overline{\mathit{cldef}}, [i]\,\texttt{new}(C\;x)?C(\overline{T}\;\overline{x}).\texttt{where}(e)\{\mathit{check}(i,e);\;\mathit{stmt};\;!\texttt{return}\}) \stackrel{\text{def}}{=} \\
\quad \mathsf{let}\;(\overline{\mathit{cldef}'}, \mathit{stmt}^{pl}) = \mathit{code}_{out}(\overline{\mathit{cldef}}, \mathit{stmt})\;\;\mathsf{in} \\
\qquad \mathsf{let}\;\;\overline{T}\;\overline{x}_p = \mathit{cparams}(C)\;\;\mathsf{in} \\
\qquad\quad \mathsf{let}\;(e', \mathit{stmt}^{pl\prime}) = (e, \mathit{stmt}^{pl})[\texttt{this}/x,\;\overline{x}_p/\overline{x}]\;\;\mathsf{in}\;\;\overline{\mathit{cldef}'}.C.C \stackrel{(i,e)}{\rightarrow} \mathit{stmt}^{pl\prime}. \\
\\
\mathit{code}_{in}(\overline{\mathit{cldef}}, \mathit{stmt}_1;\;\mathit{stmt}_2) \stackrel{\text{def}}{=} \\
\quad \mathsf{let}\;\;\overline{\mathit{cldef}_1} = \mathit{code}_{in}(\overline{\mathit{cldef}}, \mathit{stmt}_1)\;\;\mathsf{in} \\
\qquad \mathsf{let}\;\;\overline{\mathit{cldef}_2} = \mathit{code}_{in}(\overline{\mathit{cldef}_1}, \mathit{stmt}_2)\;\;\mathsf{in}\;\;\overline{\mathit{cldef}_2}. \\
\\
\mathit{code}_{in}(\overline{\mathit{cldef}}, \texttt{if}(e)\;\{\mathit{stmt}_1\}\;\texttt{else}\;\{\mathit{stmt}_2\}) \stackrel{\text{def}}{=} \\
\quad \mathsf{let}\;\;\overline{\mathit{cldef}_1} = \mathit{code}_{in}(\overline{\mathit{cldef}}, \mathit{stmt}_1)\;\;\mathsf{in} \\
\qquad \mathsf{let}\;\;\overline{\mathit{cldef}_2} = \mathit{code}_{in}(\overline{\mathit{cldef}_1}, \mathit{stmt}_2)\;\;\mathsf{in}\;\;\overline{\mathit{cldef}_2}. \\
\\
\mathit{code}_{in}(\overline{\mathit{cldef}}, \texttt{while}(e)\;\{\mathit{stmt}\}) \stackrel{\text{def}}{=} \\
\quad \mathsf{let}\;\;\overline{\mathit{cldef}_1} = \mathit{code}_{in}(\overline{\mathit{cldef}}, \mathit{stmt})\;\;\mathsf{in}\;\overline{\mathit{cldef}_1}. \\
\\
\mathit{code}_{in}(\overline{\mathit{cldef}}, \texttt{case}\;\{\;\mathit{stmt}_1\;\;\mathit{stmt}_2\;\;\ldots\;\;\mathit{stmt}_n\}) \stackrel{\text{def}}{=} \\
\quad \mathsf{let}\;\;\overline{\mathit{cldef}_1} = \mathit{code}_{in}(\overline{\mathit{cldef}}, \mathit{stmt}_1)\;\;\mathsf{in} \\
\qquad \ddots \\
\qquad \mathsf{let}\;\;\overline{\mathit{cldef}_n} = \mathit{code}_{in}(\overline{\mathit{cldef}_{n-1}}, \mathit{stmt}_n)\;\;\mathsf{in}\;\;\overline{\mathit{cldef}_n}.
\end{array}
$$

Table 4.6: Generation of *Japl* code (code_{in})

As for the transformation of a case statement, for instance, the branches are transformed subsequently, such that one branch uses the updated class definitions of the previous transformation.

4.3 Generation of the test program.

In the previous section we introduced the algorithm for generating class definitions from a given specification statement. Let us now summarize and complete the necessary steps for generating a complete test program from a specification program. Assuming that we have a valid specification program

$$s = \overline{\mathit{cutdecl}}\;\overline{T}\;\overline{x};\;\;\overline{\mathit{mokdecl}}\;\{\mathit{stmt};\;\texttt{return}\},$$

such that $\Delta \vdash s : \Theta$ for some name contexts Δ and Θ, we can generate a corresponding test program in the following way:

1. We preprocess the specification s according to Definition 4.1.4 which results in a new specification

$$s' = prep(s) = \overline{cutdecl}\ \overline{T'}\ \overline{x'};\ \overline{mokdecl}\ \{stmt';\ \texttt{return}\},$$

   which is, in particular, equipped with the anticipation code and which is free of local variables.

2. Now we translate the sequence $\overline{cutdecl}$ into an import declaration sequence $\overline{impdecl}$. To this end, each declaration `test` C defined in $\overline{cutdecl}$ is translated to `import` C, that is, we only have to replace the keyword `test` by the keyword `import`.

3. For each class definition of $\overline{mokdecl}$ we define an initial class definition with method and constructor code as given in Table 4.3 respecting the parameter and return types of the corresponding class. This results in an initial sequence of class definitions $\overline{cldef}_0$. If $stmt'$ is a passive statement we define

$$\overline{cldef} = code_{in}(\overline{cldef}_0, stmt') \quad \text{and } stmt_{pl} = \epsilon,$$

   and otherwise we define

$$(\overline{cldef}, stmt_{pl}) = code_{out}(\overline{cldef}_0, stmt').$$

4. The resulting test program is defined by

$$p = \overline{impdecl};\ \overline{T'}\ \overline{x'};\ \overline{cldef}; \{stmt_{pl};\ \texttt{return}\}.$$

4.4 Correctness of the code generation

The programming language, the test specification language, and the code generation algorithm are given in terms of formal definitions. This allows us to formally prove the correctness of the code generation algorithm. Although the language represents a relatively small subset of *Java* or C♯ the correctness proof turns out to be quite complex already. While the complete proof is given in the appendix, this section provides a discussion of the proof idea. After introducing some fundamentals regarding correctness proofs in general we will point out some specific characteristics of the test code generation. Based on this, we will outline the proof with references to the corresponding details in the appendix.

Before we deal with the actual correctness proof, we should first clarify the meaning of correctness in this context. Correctness of an algorithm in general is always to be understood with respect to a specific *specification*. That is, an algorithm is considered as correct if it meets its specification. Usually, the specification of an algorithm captures its functional aspects only, such that the specification stipulates a desired relation between an *input* to the algorithm and its generated *output*. As for our code generation algorithm, its input values are test specifications of the test specification language and its corresponding output values are

represented by the generated *Japl* test programs. Intuitively, the desired input-output relation between a test specification and the resulting *Japl* program is clear, as well:

Algorithm specification (informal): For each valid test specification s, the *Japl* program p, generated by the algorithm, has to test whether the component's behavior exposed to its environment conforms to the behavior specified by s. This has two aspects:

1. The generated test program p has to provide a proper environment for the component under test. In particular, it must not prevent a specification-conform component from showing the desired behavior.

2. Program p must detect undesired behavior.

For a formal correctness proof we likewise need a formal algorithm specification too. To this end, we have to bring the informal algorithm specification into the context of the formal language and algorithm definitions. Recall that the trace semantics of a *Japl* component consists of communication traces, where each trace captures, both, the behavior of the component exposed to its environment but also the behavior of the environment exposed to the component. Correspondingly, we defined the test specification language basically as an extension of the programming language, such that a specification's *trace semantics* serves as a description of a desired component's behavior to be exposed to its environment if reciprocally the environment exposes a certain behavior to the component. Thus, in our setting the first requirement of the informal specification above can be formalized in terms of a trace inclusion. For, each trace of the specification represents a valid behavior of the component which, therefore, must be realizable by the generated program as well. Otherwise it would prevent a specification-conform component from showing the desired behavior. Moreover, the trace inclusion ensures that the test program provides a proper environment in that it exposes the specified behavior to the component under test.

Requirement 1 (Provide a proper environment): For each well-typed specification s with $\Delta \vdash s : \Theta$ the generated test program p must have the following property:

$$[\![\Delta \vdash s : \Theta]\!] \subseteq [\![\Delta \vdash p : \Theta]\!],$$

This means that the test program may behave in the same way as the specification in that the test program *simulates* the specification. Indeed, originally introduced by Milner in [47] as a means to compare programs, simulation has become a standard proof technique for correctness proofs.[6] For systems that are given in terms of a labeled transition systems the notion of simulation is commonly defined as follows.

Definition 4.4.1 (Simulation)**:** Assume a labeled transition system $(\mathit{Conf}, a, \rightarrow)$. A *simulation relation* is a binary relation $S \in \mathit{Conf} \times \mathit{Conf}$ such that for each pair of configurations $c, d \in \mathit{Conf}$ the following holds: if $(c, d) \in S$ then for all $c' \in \mathit{Conf}$ and for all

[6] For a detailed discussion of simulation relations see also [48].

transition labels a

$$c \xrightarrow{a} c'$$

implies that there is a $d' \in Conf$ such that, using the same label a, also

$$d \xrightarrow{a} d' \quad \text{and} \quad (c', d') \in S.$$

Given two configurations c and d, we say d *simulates* c if there is a simulation S such that $(c, d) \in S$.

Thus, intuitively, a configuration d simulates another configuration c, if all the behavior that can be shown by c can also be shown by d such that d's successor again simulates the successor of c. If we relate this to the execution of the generated test program this means that, indeed, the test program must be able to realize each communication trace that is realized by the specification as well. The advantage of the simulation definition given in Definition 4.4.1 is that the trace inclusion is broken down to single transition steps only.

The definition of simulation that we gave above, however, requires that all transitions are observable, i.e., all transitions are labeled. According to the operational semantics of the specification and the programming language, in contrast, we distinguish external, i.e., labeled, from internal, i.e., unlabeled, transitions. Specifically, as for our testing approach, the generated test program need to simulate the *interface communication* of the specification only, because they represent the desired observable behavior. But we don't have to be so strict regarding the *internal transitions*. Hence, we need a slightly more relaxed simulation definition, called *weak simulation*.

Definition 4.4.2 (Weak simulation)**:** Assume a labeled transition system

$$(Conf, a, \rightarrow)$$

which also allows for unlabeled transitions. A *weak simulation relation* is a binary relation $S \in Conf \times Conf$ such that for each pair of configurations $c, d \in Conf$ the following holds:

1. if $(c, d) \in S$ then $c' \in Conf$ with

   $$c \rightsquigarrow c'$$

   implies that there is a $d' \in Conf$ such that

   $$d \rightsquigarrow^* d' \quad \text{and} \quad (c', d') \in S.$$

2. if $(c, d) \in S$ then $c' \in Conf$ and a transition labels a with

   $$c \xrightarrow{a} c'$$

   implies that there is a $d' \in Conf$ such that, using the same label a, also

   $$d \Longrightarrow^{a} d' \quad \text{and} \quad (c', d') \in S.$$

Given two configurations c and d, we say d *weakly simulates* c if there is a simulation S such that $(c, d) \in S$.

Note, in the implication of the definition's first requirement we used the star annotated internal transition arrow ($\rightsquigarrow^*$) for the transition from d to d' allowing for more than one internal transition steps but it also includes the case where d equals d'. Furthermore, the double arrow ($\overset{a}{\Longrightarrow}$) in the implication of the second requirement states that the overall transition from d to d' may consist not only of the transition step labeled with a but it may be preceded and followed by a sequence of internal transitions.

As for our code generation algorithm, the generated program p must be able to weakly simulate the specification: it must be able to produce the same observable behavior in terms of sequences of interface interactions but in between of these interactions it may perform different internal computation steps. But, intuitively, the generated code should not only support the behavior that is given by the specification but beyond that it must not support any additional behavior. This is in general captured by the notion of *bisimulation*. Bisimulation has been introduced by Park [54] for testing observational equivalence of the calculus of communicating systems. A simulation relation S is a bisimulation, if the inverse relation S^{-1} is a simulation relation as well. An equivalent definition is given in the following.

Definition 4.4.3 (Bisimulation)**:** A binary relation $S \in \mathit{Conf} \times \mathit{Conf}$ is a *bisimulation* if for all pairs of configurations $c, d \in \mathit{Conf}$ the following holds:

If $(c, d) \in S$ then for all transition labels a it is:

1. For all $c' \in \mathit{Conf}$
$$c \xrightarrow{a} c'$$
implies that there is a $d' \in S$ such that, regarding the same label a,
$$d \xrightarrow{a} d' \quad \text{and} \quad (c', d') \in S,$$

2. and, symmetrically, for all $d' \in S$
$$d \xrightarrow{a} d'$$
implies that there is a $c' \in S$ such that, regarding the same label a,
$$c \xrightarrow{a} c' \quad \text{and} \quad (c', d') \in S,$$

Given two configurations c and d in S, c is *bisimilar* to d, written $c \sim d$, if there is a bisimulation S such that $(c, d) \in S$.

The bisimilarity relation is the largest bisimulation relation of the given labeled transition system. Note, the bisimilarity relation $\sim$ is an equivalence relation. In particular, if c is bisimilar to d then d is also bisimilar to c. Note, moreover, that two configurations are not necessarily bisimilar if one configuration simulates the

other and vice versa, but instead it is important that they simulate each other regarding the same simulation relation.

Corresponding to the simulation relation, we can define a weak bisimulation which also allows internal steps.

Definition 4.4.4 (Weak bisimulation)**:** A binary relation $S \in Conf \times Conf$ is a *bisimulation* if for all pairs of configurations $c, d \in Conf$ the following holds:

Assume $(c, d) \in S$.

1. For all $c' \in Conf$ we have:

   (a) If
   $$c \rightsquigarrow c'$$
   then there is a $d' \in Conf$ such that
   $$d \rightsquigarrow^* d' \quad \text{and} \quad (c', d') \in S.$$

   (b) If, for some communication label a,
   $$c \xrightarrow{a} c'$$
   then there is a $d' \in Conf$ such that, regarding the same label a,
   $$d \stackrel{a}{\Longrightarrow} d' \quad \text{and} \quad (c', d') \in S.$$

2. Symmetrically, for all $d' \in Conf$ we have:

   (a) If
   $$d \rightsquigarrow d'$$
   then there is a $c' \in Conf$ such that
   $$c \rightsquigarrow^* c' \quad \text{and} \quad (c', d') \in S.$$

   (b) If, for some communication label a,
   $$d \xrightarrow{a} d'$$
   then there is a $c' \in Conf$ such that, regarding the same label a,
   $$c \stackrel{a}{\Longrightarrow} c' \quad \text{and} \quad (c', d') \in S.$$

Given two configurations c and d in S, c is *weakly bisimilar* to d, written $c \approx d$, if there is a weak bisimulation S such that $(c, d) \in S$.

Note, that also a weak bisimulation relation is an equivalence relation. Hence, c is weakly bisimilar to d exactly if d is weakly bisimilar to c.

Be it as it may, it is important to understand, that the generated test program is in fact *not* (weakly) bisimilar to the specification. This is due to a crucial

difference between a test and a specification: while a specification describes only the *desired* behavior of the component, a test, in contrast, has additionally to reckon with components that do not conform to the specification. That is, a test can fail. This is reflected by the fact that the test program's trace semantics also includes traces which entail an undesired behavior of the component under test. In particular, the trace semantics of the generated program is not equal to the trace semantics of the test specification. However, the test program should detect an undesired behavior of the component as soon as possible and react with a failure report. Correspondingly, a trace of the generated test program may only deviate from the specification due to an undesired behavior caused by the component under test but the observable behavior of the test program itself must be always as specified. As a consequence, the second requirement for a correct code generation algorithm cannot be expressed by a simple trace inclusion statement but it has to account for the possibility of undesired incoming communication. This can be formalized as follows.

Requirement 2 (Detect undesired behavior):

1. $s\gamma! \in [\![\Delta \vdash p : \Theta]\!]$ implies $s\gamma! \in [\![\Delta \vdash s : \Theta]\!]$
2. $s\gamma? \in [\![\Delta \vdash p : \Theta]\!]$ implies either $s\gamma? \in [\![\Delta \vdash s : \Theta]\!]$
   or $\Delta \vdash c_{init}(p) : \Theta \overset{s\gamma?}{\Longrightarrow}\downarrow_{\mathsf{fault}}$.

We said that intuitively the test program should support the specified interface behavior – but nothing more. Indeed, the second requirement for a correct code generation algorithm resembles the first requirement, in that the trace inclusion is split into two parts regarding the last communication label — the first part does demand trace inclusion and only the second part adds some extra behavior: All traces of the program that end with an outgoing communication must be included in the specification's trace semantics, as well. Traces of the program that end with an incoming communication, however, must *either* be included in the specification's trace semantics *or* otherwise the program must report a failure after realizing the trace. For, the latter case represents a program execution where the last interface communication due to the component under test was not expected according to the specification.

Thus, as mentioned above already, the second requirement cannot be formulated in terms of the usual simulation relation but we have to come up with a similar yet slightly different definition.

Definition 4.4.5 (Testing simulation)**:** Assume a labeled transition system

$$(Conf, a, \rightarrow)$$

which also allows for unlabeled transitions. A *testing simulation relation* is a binary relation $S \in Conf \times Conf$ such that for each pair of configurations $c, d \in Conf$ the following holds:

1. if $(c, d) \in S$ then $c' \in Conf$ with

$$c \rightsquigarrow c'$$

implies that there is a $d' \in Conf$ such that

$$d \rightsquigarrow^* d' \quad \text{and} \quad (c', d') \in S.$$

2. if $(c, d) \in S$ then $c' \in Conf$ and a transition labels a with

$$c \xrightarrow{a} c'$$

implies that

(a) either there is a $d' \in Conf$ such that, using the same label a, also

$$d \stackrel{a}{\Longrightarrow} d' \quad \text{and} \quad (c', d') \in S$$

(b) or otherwise there exist no such $d' \in Conf$ but instead

$$c' \downarrow_{\mathsf{fault}} .$$

Given two configurations c and d, we say d *simulates* c *up to test failures* if there is a testing simulation S such that $(c, d) \in S$.

As a consequence, the desired input output relation for our code generation algorithm is not captured by a bisimulation relation but we have to combine the simulation aspect with the testing simulation.

Definition 4.4.6 (Testing bisimulation)**:** A simulation relation $S \in Conf \times Conf$ is a *testing bisimulation* if S^{-1} is a testing simulation. Given two configurations c and d, we say d is *testing bisimilar* to c (or: d is weakly bisimilar to c *up to testing failures*), written $c \precsim d$, if there exists a testing bisimulation S with $(c, d) \in S$.

Note, in contrast to bisimulation, testing bisimulation is not symmetric: the generated test program simulates the specification, but the specification simulates the test program up to test failures only.

Summarizing, we state the correctness of the generated test program and we subsequently sketch the proof.

Lemma 4.4.7 (Correctness of the test program generation)**:** Let s be a well-typed test specification and, correspondingly, let p be the *Japl* program that results from s according to the test program generation algorithm given in Section 4.3. Then p fulfills Requirement 1 and Requirement 2.

The complete proof of Lemma 4.4.7 is given in the appendix. Yet, in the following we present the general proof idea. We have to show that for each specification s and for the correspondingly generated program p we can provide a relation S such that S represents a testing bisimulation for s and p. More precisely, we have defined the different simulation and bisimulation relations for configurations, only.

Therefore, we actually have to provide a testing bisimulation relation R_b which includes the pair consisting of the initial configurations regarding s and p, respectively, i.e.,

$$(c_{init}(s), c_{init}(p)) \in R_b.$$

For the sake of brevity, however, we will use in this context the specification and the program as a shorter representation for their initial configurations, such that we can also write

$$(s, p) \in R_b.$$

Since the code generation algorithm is given in two parts, namely the preprocessing step and the actual *Japl* code generation step, we will break down the proof into two parts, as well.

The preprocessing step results in a new specification which must not show a different behavior than the original specification. In particular, it does not entail the above mentioned specification-test discrepancy but the result of the preprocessing step still represents a specification. Therefore, we have to show that the original specification and the preprocessed specification are indeed (weakly) bisimilar. Regarding the actual *Japl* code generation step, however, we have to deal with the discrepancy between a specification and a test. Thus, we have to show that the preprocessed specification and the resulting *Japl* code are related with respect to a testing bisimulation. The combination of both proofs yield the result that the original specification and the *Japl* code are testing bisimilar. The correctness proof of the code generation algorithm can be sketched as follows. For a given specification s we first provide a weak bisimulation relation R_b with $(s, prep(s)) \in R_b$ where $prep(s)$ is the specification that results from preprocessing s. Second, we give a testing bisimulation relation R_t with $(prep(s), p) \in R_t$ where p is the *Japl* program that results from generating *Japl* code from $prep(s)$. Thus, we will prove

$$s \approx prep(s) \precsim p$$

for each input output pair, s and p of the code generation algorithm.

Section C.1 of the appendix deals with the bisimulation proof regarding the preprocessing step, i.e., we prove $s \approx prep(s)$. More specifically, the proof of Lemma C.1.3 shows that the specification which results from applying the preprocessing functions $prep_{in}$ and $prep_{out}$, defined in Section 4.1, is bisimilar to the original specification. In fact, the proof does not take the variable globalization step into account. However, due to the absence of recursion it is obvious that the variable globalization does not affect the observable behavior of the specification, hence, we can derive from the proof that a specification s and its preprocessed version $prep(s)$ are bisimilar.

Next, we have to prove that $prep(s) \precsim p$ where p represents the *Japl* program which has been generated from $prep(s)$ according to Section 4.2 and 4.3. This is subject to Section C.3. Regarding this testing bisimilarity proof, two complications arise. First, the operational semantics of the *Japl* language is formalized in context of a specific program p. For instance, Rule Call refers to the implementation of

the corresponding method in p. As a consequence, we have to include the program p into the definition of the testing bisimulation relation. More specifically, for each program p we provide a corresponding relation R_b^p.

Second, as mentioned in Section 4.1, the transition from a specification s to a corresponding *Japl* program p renders it necessary to distribute the originally sequential specification over several method bodies and classes. On the other hand, testing bisimilarity certainly entails the requirement that p sticks to the originally specified order of interactions. This is were the anticipation mechanism comes into play. That is, in Section C.1 we prove that already the preprocessed specification is equipped with properly anticipating code regarding the next incoming communication step.

4.5 Failure report and faulty specifications

Up to this point, we assumed that a test program just stops execution, if the unit under test shows an undesired behavior. More precisely, we assumed that the abstract code *check(i,e)* and *assert(e)* diverge, if it detects an unexpected incoming communication. This was sufficient for the theoretical considerations so far, but in practice such a reaction certainly isn't very helpful. Instead, a test program should report if the behavior of the unit under test does not comply with the specification. To this end, we assume an additional external component which allows for printing error messages. If an expectation identifier check or a where-clause assertion fails, the external component is used to report a failed test run. Afterwards, since the language does not provide a statement for an abnormal termination, we can stop the program with an infinite empty while-loop.

Clearly a test run fails if the unit under test implements an interface communication which is unexpected according to the specification. But what are actually the criteria for passing a test? A straightforward and intuitive criterion for a successful test run would be a test program execution that reaches a terminal configuration such that the corresponding interface trace is also an element of the specification's trace semantics. Due to while-loops, however, a specification can specify desired interface traces of arbitrary length. This allows to specify and test the interface behavior of *reactive systems* which usually do not terminate with a final result but are expected to interact with their environment continuously. Thus, not only a test run that ended in a terminal configuration is considered to be successful but also all (possibly ongoing) executions — unless the unit under test shows an undesired interface communication. The only trivial difference between a successful terminated test run and an ongoing test run is that a ongoing successful test run can still become a faulty test run due to an undesired behavior of the unit, whereas the terminated successful test run cannot become faulty anymore. In cases where we want to emphasis the latter kind of successful test runs we speak of an *irrevocably successful test run.*

In classic state-based testing usually only terminating test runs can be assessed. For, at the very end of the test program, a test verdict is derived from the final program state. It wouldn't make sense to say anything about test suc-

cess or failure for an ongoing test execution. In our testing approach, however, a test verdict is spread on the complete interface trace: each occurrence of an expected incoming communication represents a desired behavior of the unit under test. Thus, we do not only increase confidence on the unit with each terminated successful test run, but we increase confidence already during a test execution. This justifies our decision to count ongoing test runs as successful test runs.

With this notion of success, we do not need to add any code for success reports: we assume that a termination of the program is visible, which indicates an irrevocably successful test run; likewise a program which is still in execution, but in absence of failure reports so far, is considered to be successful.

Nevertheless, again in the real world, this notion of success is also not always useful. A diverging unit would be considered as successful, although this seldom complies with the desired behavior. On account of this, we additionally assume that the system provides the possibility to implement a time-out, such that an expected incoming communication has to occur within a certain time period, otherwise the test program reports a *time-out failure* and stops. In a simpler setting, the test program continuously reports on the progress and the software tester can decide to stop the program if he or she notices that the program hasn't made any progress for a longer time.

Note, that as in other testing approaches, a failed test execution does not necessarily result from a faulty unit but it also may indicate a *faulty specification*. To understand this, consider the following specification snippet.

Listing 4.8: Faulty traces: specification snippet

```
1 count = 1; lastval = 1000;
2 while(count <= 10) {
3   (C x)?meth(int i).where(i > 0 && i < lastval) {
4     count = count + 1;
5     lastval = i;
6     !return(null)
7   }
8 }
```

In the example, a while-loop expresses the expectation of 10 consecutive incoming calls, all addressing the same class and method. Moreover, a global variable *lastval* is used within the call statement's where-clause in order to assure that the value of the call's parameter is greater than zero and less than the parameter's value of the last call or, in case of the first call, less than the initial value of 1000.

Now, let us assume a deduction in the operational semantics of the specification language where the parameter of the first incoming call is 5. Although this call is satisfying the where-clause it makes, at the same time, a complete processing of the specification program impossible, since for each of the expected consecutive calls the value of the parameter has to be decreased at least by one. Thus, the deduction gets stuck meaning that the specification program cannot reach the terminal configuration. However, the first call could have had a different value, such that in this case finishing the program would have been possible.

In other words, the feasibility for reaching a terminal configuration depends on the behavior of the external component. For this reason, we consider the behavior shown by the external component of our example as incorrect.

However, if we modify the specification example such that the initial value of the variable *lastval* is not 1000 but 5, then things are different. In this case, no matter what the incoming behavior is, it is not possible to reach a terminal configuration. Therefore, we call the modified specification a *faulty specification*. A trace of a specification which cannot be extended by further communication steps due to unsatisfiable where-clauses is called a *faulty trace*. Note, that a single trace can be faulty due to a faulty specification or due to an incorrect behavior of the external component. A specification is exactly a faulty specification if all of its traces are faulty traces.

Note, that also faulty specifications are satisfiable, as the operational semantics of a faulty specification cannot produce a matching incoming communication label, either. In particular, also regarding the specification language, a faulty specification can never reach a terminal configuration.

Chapter 5

Further possible extensions

Within the last chapters we have provided a basic framework for testing components of object-oriented class-based languages like *Java* and C♯. A main contribution was the development of a test specification language which allows to specify a desired interface trace in order to stipulate the expected interface behavior of an object-oriented component under test. As mentioned earlier, however, some of the common features of object-oriented programming languages have been omitted. In this chapter we want to discuss some of these features. In particular, we sketch possible approaches to incorporating certain features into our programming language. We also investigate the extension's impact on our testing approach and correspondingly suggest additional modifications of the specification language, if necessary. Furthermore, we discuss some extensions concerning the specification language only, that is, language features that may facilitate writing test specification.

5.1 Specification classes

We have introduced a test specification language which can be used to describe expected interface interactions of communicating *objects*. The specification language itself, however, is not object-oriented. Extending the specification language with classes and objects may allow for reusing and parameterizing specifications.

Specifically, in this section we want to investigate an extension of the specification language with *specification classes*. Method bodies of specification classes consist of specification statements. An invocation of such a specification method gives rise to the expectation of the interface interaction sequence given by the method's body. A specification statement within a method body might contain reference to fields and to parameters of the method as well as calls to other specification methods. In particular, a specification method may call itself, i.e., the extension of the language introduces recursion. Summarizing, a method body of

s ::=	$\overline{cutdecl}\ \overline{T}\,\overline{x};\ \overline{mokdecl}\ \overline{cldef}\ \{\ stmt\ \}$	specification
$cutdecl$ ::=	$\texttt{test class}\ C;$	test unit class
$mokdecl$ ::=	$\texttt{mock class}\ C\{C(T,\ldots,T);\ \overline{T\ m(T,\ldots,T)}\};$	mock class
$cldef$ ::=	$\texttt{class}\ C\{\overline{T}\,\overline{f};\ con\ \overline{mdef}\}$	class def.
con ::=	$C(\overline{T}\,\overline{x})\{\overline{T}\,\overline{x};\ stmt;\ \texttt{return}\}$	constructor
$mdef$ ::=	$X\ m(\overline{T}\,\overline{x})\{\overline{T}\,\overline{x};\ stmt;\ \texttt{return}\}$	meth. def.
$stmt$ ::=	$x = e \mid x = \texttt{new}\ C() \mid \varepsilon \mid stmt; stmt \mid \{\overline{T}\,\overline{x};\ stmt\}$	statements
	$\mid \texttt{while}\ (e)\ \{stmt\} \mid \texttt{if}\ (e)\ \{stmt\}\ \texttt{else}\ \{stmt\}$	
	$\mid stmt_{in} \mid stmt_{out} \mid \texttt{case}\ \{\ \overline{stmt_{in}; stmt}\ \}$	
	$\mid f = e \mid e.m(e,\ldots,e) \mid x = \texttt{new}\ C(e,\ldots,e)$	
$stmt_{in}$ ::=	$(C\ x)?m(\overline{T}\,\overline{x}).\texttt{where}(e)\ \{\overline{T}\,\overline{x};\ stmt;\ !\texttt{return}\ e\}$	incoming stmt
	$\mid \texttt{new}(C\ x)?C(\overline{T}\,\overline{x}).\texttt{where}(e)\ \{\overline{T}\,\overline{x};\ stmt;\ !\texttt{return}\}$	
$stmt_{out}$ ::=	$e!m(e,\ldots,e)\ \{\overline{T}\,\overline{x};\ stmt;\ ?\texttt{return}(x).\texttt{where}(e)\}$	outgoing stmt
	$\mid \texttt{new}!C(e,\ldots,e)\ \{\overline{T}\,\overline{x};\ stmt;\ ?\texttt{return}(x).\texttt{where}(e)\}$	
e ::=	$x \mid f \mid \texttt{this} \mid \texttt{null} \mid \texttt{op}(e,\ldots,e)$	expressions
X ::=	$? \mid !$	control context

Table 5.1: Extension by specification classes: syntax

a specification class represents a trace specification which is possibly abstracted over parameters, variables, and sub-traces.

Note that introducing specification classes, renders it necessary to distinguish them from mock classes. While mock classes are still given in terms of their signature only, specification classes in contrast are fully-fletched classes similar to the programming language classes except that their method bodies may contain expectation statements. Furthermore, mock objects still neither provide fields nor do they allow for internal method calls. On the contrary, specification classes and their objects must not show up at the interface trace, hence, they can be considered as hidden classes with respect to the specification program's environment. We will capture these requirements by the type system, which we will explain somewhat later.

Table 5.1 suggests a grammar extension of the specification language regarding specification classes. We have extended the grammar of the original specification language given in Table 3.1 by constructs for class definitions and by statements for method and constructor calls as well as field updates. The definition of specification classes resembles the definition of conventional classes in the programming language but differs in two aspects. First, for simplicity reasons, method definitions do not include return values and therefore not a return type either. Second, instead of a return type, method definitions state the control context X of the body statement. Note, that a method call statement does not entail an assignment, due to the lack of a return value. Finally, we have added rules for expressions that yield the current object's name or the value of one of its fields, respectively.

We said earlier, that we have to distinguish mock and specification classes. The proper differentiation will be carried out by the type system. In particular, internal method calls as well as field accesses may only be targeted at instances of specification classes, while interface communication statements may only involve external or mock classes.

Additionally, we have to ensure that the new constructs do not allow to specify an inconsistent control flow. For, in general we want to allow invocations of specification methods to appear within, both, passive and active control context, yet we have to check that a specification method's body complies with the control context of its call. In account of this, we extend the type definition given in Definition 2.2.1 by adding the rule

$$T ::= (MNames \cup CNames) \rightharpoonup (U \times \ldots \times U)^{\gamma}.$$

The new type is used for specification classes. It yields the parameter types and the control context γ of each of the specification class's methods and constructor, allowing to check for control flow consistency. At the same time, it also allows to distinguish mock and specification classes, as mock classes will be associated with the usual class types.

The type system of the original specification language is extended by rules for the new constructs. These new rules resemble the corresponding typing rules of the programming language as given in Table 2.2 and Table 2.3. Besides adding new rules we only have to modify the Rule T-Spec in order to deal with the new class definitions. Thus, Table 5.2 only shows the new version of Rule T-Spec as well as the new rules concerning the new class definition constructs and the new statements. All other rules that were given in Table 3.2 are inherited without any modifications and are, therefore, left out. We also omit the straightforward typing rules for the new expressions.

As mentioned earlier, we extend Rule T-Spec with a judgment for type checking the specification class definitions. Additionally, we have to ensure that specification types do not show up in the interface communication. That is, the signatures of methods and constructors of both, mock classes and imported classes, must not include class names of specification classes. This check is abbreviated by the new premise $\Delta \vdash \Theta : \mathsf{ok}$, which stands for:

$$\begin{aligned}
&\forall C \in dom(\Delta).\ \forall C' \in \mathsf{commedCl}(C, \Delta).\ C' \in dom(\Theta) \Rightarrow \mathsf{isMockCl}(C') \\
&\wedge \\
&\forall C \in dom(\Theta).\ \mathsf{isMockCl}(C) \Rightarrow \forall C' \in \mathsf{commedCl}(C, \Theta). \\
&\quad C' \in dom(\Theta) \Rightarrow \mathsf{isMockCl}(C'),
\end{aligned}$$

where we use the following two auxiliary functions in order to determine the set of communicated class names within the signature of a given class and to find out if a class is a mock class but not a specification class:

$$\mathsf{commedCl}(C, \chi) \stackrel{\text{def}}{=} \cup_{m \in dom(C)} \{\chi(C)(m).dom \cup \chi(C)(m).ran\} \quad \text{and}$$

$$\mathsf{isMockCl}(C) \stackrel{\text{def}}{=} \Theta(C) \in (FNames \cup CNames) \rightharpoonup (T \times \ldots \times T \rightarrow T)$$

$$[\text{T-Spec}]\ \frac{\Gamma;\Delta \vdash \overline{cutdecl} : \mathsf{ok} \quad \Theta = cltype(\overline{mokdecl}), cltype(\overline{cldef}) \quad \Delta \vdash \Theta : \mathsf{ok} \quad \Gamma' = \Gamma, \overline{x}{:}\overline{T} \quad \Gamma';\Delta;\Theta \vdash \overline{cldef} : \mathsf{ok} \quad \Gamma';\Delta;\Theta \vdash stmt : \mathsf{ok}^{\gamma}}{\Gamma;\Delta \vdash \overline{cutdecl}\ \overline{mokdecl}\ \overline{T}\ \overline{x};\ \overline{cldef}\ \{\ stmt\ \} : \Theta^{\gamma}}$$

$$[\text{T-SClass}]\ \frac{\Gamma' = \Gamma, \overline{f}{:}\overline{T}, \mathtt{this}{:}C \qquad \Gamma';\Delta;\Theta \vdash con : \mathsf{ok} \qquad \Gamma';\Delta;\Theta \vdash \overline{mdef} : \mathsf{ok}}{\Gamma;\Delta;\Theta \vdash \mathtt{class}\ C\{\overline{T}\ \overline{f};\ con\ \overline{mdef}\} : \mathsf{ok}}$$

$$[\text{T-Con}]\ \frac{\Gamma' = \Gamma, \overline{x}{:}\overline{T}, \overline{x'}{:}\overline{T'} \qquad \Gamma';\Delta;\Theta \vdash stmt : \mathsf{ok}^{act}}{\Gamma;\Delta;\Theta \vdash C(\overline{T}\ \overline{x})\{\overline{T'}\ \overline{x'};\ stmt;\ \mathtt{return}\} : \mathsf{ok}}$$

$$[\text{T-MDef}]\ \frac{\Gamma' = \Gamma, \overline{x}{:}\overline{T}, \overline{x'}{:}\overline{T'} \qquad \Gamma';\Delta;\Theta \vdash stmt : \mathsf{ok}^{\gamma(X)}}{\Gamma;\Delta;\Theta \vdash X\ m(\overline{T}\ \overline{x})\{\overline{T'}\ \overline{x'};\ stmt;\ \mathtt{return}\} : \mathsf{ok}}$$

$$[\text{T-Call}_{\mathsf{SCl}}]\ \frac{\Gamma;\Delta,\Theta \vdash e : C \quad \Theta(C)(m) = \overline{T}^{\gamma} \quad \Gamma;\Delta \vdash \overline{e} : \overline{T}}{\Gamma;\Delta;\Theta \vdash e.m(\overline{e}) : \mathsf{ok}^{\gamma}}$$

$$[\text{T-New}_{\mathsf{SCl}}]\ \frac{\Theta(C)(C) = \overline{T}^{act} \quad \Gamma;\Delta,\Theta \vdash \overline{e} : \overline{T}}{\Gamma;\Delta;\Theta \vdash x = \mathtt{new}\ C(\overline{e}) : \mathsf{ok}^{act}} \qquad [\text{T-FUpd}_{\mathsf{SCl}}]\ \frac{\Gamma;\Delta,\Theta \vdash e : \Gamma(f)}{\Gamma;\Delta;\Theta \vdash f = e : \mathsf{ok}^{act}}$$

Table 5.2: Extension by specification classes: type system (stmts)

In words, each externally defined class shall communicate only instances of tester classes that are indeed mock classes. Likewise, each mock class shall only communicate instances of tester classes that are mock classes, too.

The new typing Rule T-SClass deals with specification classes and is almost identical to Rule T-Class for programming classes except that the assumed typing context is conformed to the typing context of the specification language's type system. Note, that a class definition is well-typed in a passive and in an active control context. Also constructor and method definitions are well-typed in any control context as can be seen from Rule T-SCon and Rule T-SMDef, respectively. The body of a constructor definition, however, is only well-typed in an active control context; a method body is type-checked in the control context that has been stated in the method definition. Consequently, an internal method call is only well-typed if it occurs in a control context which corresponds to the control context of the called method (T-CallSCl). This check also ensures that no mock method can be called internally, as mock classes do not provide control contexts for their methods at all. An instantiation of a specification class is handled in Rule T-NewSCl. It may only occur in an active control context as it involves a side-effect in form of a variable update. For the same reason, also field updates are only allowed in an active control context, as can be seen in Rule T-FUpdSCl.

Concerning the operational semantics, we can leave the rules regarding the interface communication as given in Table 3.3 in Section 3.4. For, the specification classes must not make any contributions to the interface communication. As for the internal computation steps, the operational semantics has to be extended by new rules for the three new statements, namely for field updates, method calls and constructor calls of specification classes. Additionally, we have to add rules for the return from internal method and constructor calls. Fortunately, we can borrow the corresponding internal rules of the programming language of Table 2.7 with almost no modifications. We only have to simplify Rule Call and Ret, since in the specification language the internal calls do not return values. The new rules are given in Table 5.3. Finally, the new constructs do not entail new types of interface communications, hence, we do not have to extend or modify the transition rules dealing with the interface communication.

$$[\text{FUpd}]\ \frac{o = [\![\texttt{this}]\!]_h^{\mathsf{v},\mu} \quad (C,\mathsf{F}) = h(o) \quad h' = h[o \mapsto (C, \mathsf{F}[f \mapsto [\![e]\!]_h^{\mathsf{v},\mu}])]}{(h, \mathsf{v}, (\mu, f = e; mc) \circ \mathsf{CS}^b) \rightsquigarrow (h', \mathsf{v}, (\mu, mc) \circ \mathsf{CS}^b)}$$

$$[\text{Call}]\ \frac{\begin{array}{c} o = [\![e]\!]_h^{\mathsf{v},\mu} \quad C = h(o).class \quad \overline{T}\,\overline{x} = mparams(C)(m) \quad \overline{T_l}\,\overline{x_l} = mvars(C)(m) \\ \mathsf{v}_l = \{\texttt{this} \mapsto o, \overline{x} \mapsto [\![\overline{e}]\!]_h^{\mathsf{v},\mu}, \overline{x_l} \mapsto ival(\overline{T_l})\} \end{array}}{(h, \mathsf{v}, (\mu, e.m(\overline{e}); mc \circ \mathsf{CS}^b) \rightsquigarrow (h, \mathsf{v}, (\mathsf{v}_l, mbody(C,m)) \circ (\mu, \texttt{rcv};\ mc) \circ \mathsf{CS}^b)}$$

$$[\text{New}]\ \frac{\begin{array}{c} o \in N \setminus dom(h) \quad h' = h[o \mapsto Obj_\perp^C] \quad \overline{T}\,\overline{x} = cparams(C) \quad \overline{T_l}\,\overline{x_l} = cvars(C) \\ \mathsf{v}_l = \{\texttt{this} \mapsto o, \overline{x} \mapsto [\![\overline{e}]\!]_h^{\mathsf{v},\mu}, \overline{x_l} \mapsto ival(\overline{T_l})\} \end{array}}{\begin{array}{l} (h, \mathsf{v}, (\mu, x = \texttt{new}\ C(\overline{e}); mc) \circ \mathsf{CS}^b) \rightsquigarrow \\ (h', \mathsf{v}, (\mathsf{v}_l, cbody(C);\ \texttt{return this}) \circ (\mu, \texttt{rcv}\ x;\ mc) \circ \mathsf{CS}^b) \end{array}}$$

$$[\text{Ret}_m]\ (h, \mathsf{v}, (\mu, \texttt{return}) \circ (\mu', \texttt{rcv};\ mc) \circ \mathsf{CS}^b) \rightsquigarrow (h, \mathsf{v}, (\mu', mc) \circ \mathsf{CS}^b)$$

$$[\text{Ret}_c]\ \frac{(\mathsf{v}', \mu'') = vupd(\mathsf{v}, \mu', x \mapsto [\![e]\!]_h^{\mathsf{v},\mu})}{(h, \mathsf{v}, (\mu, \texttt{return}\ e) \circ (\mu', \texttt{rcv}\ x;\ mc) \circ \mathsf{CS}^b) \rightsquigarrow (h, \mathsf{v}', (\mu'', mc) \circ \mathsf{CS}^b)}$$

Table 5.3: Extension by specification classes: operational semantics

Although extending the specification language by specification classes was more or less straightforward, the code generation algorithm gets considerably more complex. This has three reasons. First, introducing recursion entails the possibility that several instances of an expectation statement's local variable exist in the variable stack at the same time, rendering it impossible to replace them by global variables. The same applies to the parameters of an expectation statement. Consider for instance a specification that contains the following method definition:

```
?specMeth(C o1) {
  o1?mockMeth(C o2, D o3){
    o3!unitMeth(){
```

```
      this.specMeth(o2);
      ?return()
    };
    !return(o3)
  };
  return
}
```

The body of the specification method *specMeth* represents the expectation of an incoming call of *mockMeth*, which is not answered with an immediate return but provokes a call-back of method *unitMeth* to the component under test. The body of the call-back specification, in turn, contains a recursive call of the specification method *specMeth*.

This example makes clear, that introducing global variables for the incoming call's parameters *o2* and *o3* is not sufficient, as the execution of the specification may lead to two instances of the incoming call statement of *mockMeth* on the call stack, where each instance needs its own parameter representation in the variable stack. As a consequence, the parameters and local variables of interaction statements cannot be replaced by global variables anymore but one has to emulate the variable stack of the called methods.

The second complication concerns the update of the global variable *next* which we used to anticipate the next incoming communication. A specification program may contain several internal call statements referring to the same specification method, such that each call statement requires a different update of *next* after the specification method has been processed. For a better understanding, consider the following specification snippet:

```
?specMeth() {
  (C x)?mockMeth1(){ !return() };
  return
}
⋮
{ // main body of specification :
  specMeth(); (D y)?mockMeth2() { !return() };
  specMeth(); (E z)?mockMeth3() { !return() }
}
```

Thus, we have defined a specification methods *specMeth*, which is called by the main body twice, such that each call is followed by the incoming call statement of another mock method (namely *mockMeth2* and *mockMeth3*, respectively). Following the labeling approach suggested in Chapter 4, we would equip the incoming call of *mockMeth2* and *mockMeth3* with expectation ids, say, i_2 and i_3. Moreover, we would have to insert an update of *next* in front of the last outgoing communication term that precedes the incoming call statements. In our example, however, both the incoming call statements, *mockMeth2* and *mockMeth3*, are preceded by the outgoing return term in the specification method *specMeth*. Thus, we cannot determine the identifier of the next expected incoming call statically, as *specMeth*

is called more than once. A solution to this problem is to adapt the preprocessing steps such that we add a parameter to each specification method for incoming communications. The parameter is used to determine the desired update statement for *next*. Thus, regarding our example, the outcome of the preprocessing could be sketched in the following way:

```
?specMeth(int updatebranch) {
  [i1](C x)?mockMeth1(){
    if (updatebranch == 1) { nxt = i2 }
    else { };
    if (updatebranch == 2) { nxt = i3 }
    else { };
    !return };
  return
}
⋮
{ // main body of specification :
  ...
  specMeth(1); [i2](D y)?mockMeth2() { ... !return };
  specMeth(2); [i3](E z)?mockMeth3() { ... !return }
 }
```

The third complication arise from the fact that specification methods for passive control contexts, i.e., a specification method whose body starts with an incoming call statement, cannot be translated to a corresponding method in the programming language. Again consider a small example

```
?specMeth(C x) {
  x?mockMeth(){ ... !return };
  return
}
⋮
{ // main body of specification :
  ... o1!unitMeth() { specMeth(o2); ... ?return }
}
```

In this example, a specification method *specMeth* is given whose body consists of an incoming call statement where the expected callee is determined by the specification method's parameter x. An invocation of method *specMeth* happens in the main body right after an *outgoing* call term. Thus, this internal call cannot be carried out in the programming language as it does not allow internal computation steps right after an outgoing communication. Moreover, we know from Chapter 4 that the incoming call statement in *specMeth* will be translated to a fragment of the method definition of method *mockMeth*. However, certainly the method *mockMeth* won't have direct access to parameters of the specification method.

To overcome these problems, we suggest the following approach. According to Rule Call in Table 5.3, the invocation of a specification method results into a

new activation $(\mathsf{v}_l, mbody(C)(m)$ which may evolve to some (μ, mc), in general. In the translated code, we emulate the call of a specification method that appears within passive control contexts by providing a global variable *specMethVars* which captures the variable function lists μ of its activation records. Since passive specification methods do not contain active statement that have to be carried out by the specification method, we do not need a representation of the activation record's code mc. The variable *specMethVars* consists of a list of structures where each structure contains the a specification method's local variable list μ including its actual parameters as well as a reference to the corresponding specification object. Accesses to the parameters and local variables of a specification method are replaced by accesses to the structure. Accesses to fields of a specification object are replaced by calls to designated access methods.

Now, a call of a passive specification method has to be anticipated such that a corresponding structure is created right before the preceding outgoing communication happens. Correspondingly, prior to the last outgoing communication term within the specification method the structure of the specification method has to be deleted. Hence, we have to extend the anticipation mechanism of Section 4.1 such that is does not only handle the anticipation of incoming call expectations but also the anticipation regarding invocations of specification methods that entail an incoming call expectation.

5.2 Programming classes

The previous section has shown that extending the specification language with specification classes requires a complex adaption of the code generation process. Extending the specification language with programming language classes, in contrast, only involves a rather moderate adaption. More specifically, we want to allow the usage of classes whose method bodies do not contain any specification statements but only programming language statements. Calls to these methods may only occur within an active control context. Instances of these classes may show up at the interface only in object position, i.e., as a parameter or return value but not as a callee. Additionally, we want to support the import of programming language classes. This facilitates writing a specification program, as it allows, for instance, to import standard library classes implementing common data structures as sets or lists or the like.

Table 5.4 shows the grammar for a specification language extended by programming language classes. Actually, the syntactical extension of the specification language is very similar to the modification of the last section. Again we borrow the class definition constructs from the grammar of the programming language but this time we don't have to adapt the original method definition but we keep the return expression and the type in the corresponding construct. We furthermore extend the specification construct with the support for import declarations.

The idea is to embed the class concept of the programming language in the specification language, such that classes of programming language components can

s ::=	$\overline{cutdecl}\ \overline{impdecl}\ \overline{mokdecl}\ \overline{T}\,\overline{x};\ \overline{cldef}\ \{\ stmt\ \}$	specification
$cutdecl$ ::=	$\texttt{test class}\ C;$	test unit class
$impdecl$ ::=	$\texttt{import}\ C;$	imported class
$mokdecl$ ::=	$\texttt{mock class}\ C\{C(T,\ldots,T);\ \overline{T\ m(T,\ldots,T)}\};$	mock class
$cldef$ ::=	$\texttt{class}\ C\{\overline{T}\,\overline{f};\ con\ \overline{mdef}\}$	class def.
con ::=	$C(\overline{T}\,\overline{x})\{\overline{T}\,\overline{x};\ stmt;\ \texttt{return}\}$	constructor
$mdef$ ::=	$T\ m(\overline{T}\,\overline{x})\{\overline{T}\,\overline{x};\ stmt;\ \texttt{return}\ e\}$	meth. def.
$stmt$ ::=	$x = e \mid x = \texttt{new}\ C() \mid \varepsilon \mid stmt; stmt \mid \{\overline{T}\,\overline{x};\ stmt\}$	statements
	$\mid \texttt{while}\ (e)\ \{stmt\} \mid \texttt{if}\ (e)\ \{stmt\}\ \texttt{else}\ \{stmt\}$	
	$\mid stmt_{in} \mid stmt_{out} \mid \texttt{case}\ \{\ \overline{stmt_{in}; stmt}\ \}$	
	$\mid f = e \mid x = e.m(e,\ldots,e) \mid x = \texttt{new}\ C(e,\ldots,e)$	
$stmt_{in}$ ::=	$(C\ x)?m(\overline{T}\,\overline{x}).\texttt{where}(e)\ \{\overline{T}\,\overline{x};\ stmt;\ !\texttt{return}\ e\}$	incoming stmt
	$\mid \texttt{new}(C\ x)?C(\overline{T}\,\overline{x}).\texttt{where}(e)\ \{\overline{T}\,\overline{x};\ stmt;\ !\texttt{return}\}$	
$stmt_{out}$ ::=	$e!m(e,\ldots,e)\ \{\overline{T}\,\overline{x};\ stmt;\ ?\texttt{return}(x).\texttt{where}(e)\}$	outgoing stmt
	$\mid \texttt{new}!C(e,\ldots,e)\ \{\overline{T}\,\overline{x};\ stmt;\ ?\texttt{return}(x).\texttt{where}(e)\}$	
e ::=	$x \mid f \mid \texttt{this} \mid \texttt{null} \mid \texttt{op}(e,\ldots,e)$	expressions

Table 5.4: Extension by programming classes: syntax

be used within specifications. Thus, it is important that class definitions which represent syntactically correct and well-typed definitions regarding the programming language are also syntax valid and well-typed regarding the specification language. Moreover, such a class definition executed within a specification should gives rise to the same semantics as in the programming language. A comparison of the extended grammar of the specification language, given in Table 5.4, with the grammar of the programming language, given in Table 2.1 and Table 2.8, shows that indeed all instances of *cldef* in the grammar of Table 2.1 are also instances of *cldef* in the grammar of Table 5.4: The grammar rules for the class, constructor, and method definitions are identical; moreover, all statements of the programming language are statements of the specification language, too. The converse, however, does not hold, since the statements in the specification language also comprise the interaction statements, which do not exist in the programming language.

We want to restrict the class definitions of the specification language to class definitions of the programming language. In particular, a class' method definitions must not contain expectation statements. This restriction has to be carried out by the type system. To this end, we introduce a new kind of control context *int*, called *internal control context*, which represents a subset of the active control context. That is, every statement which is considered to occur in internal control context is also in active control context. A statement is in internal control context if the specification has the control and if the statement is also represented in the syntax of the programming language. In particular, outgoing call statements may occur in active control context but never in internal control context. Finally, in

order to restrict the class definitions to the desired ones, the type system will allow method body to contain only statements which are in internal control context.

Furthermore, the type system has to ensure that the new programming language classes do not occur as callee in specification statements. This is done by introducing another typing context, Π, which includes all typed programming language classes. Specifically, it contains the locally defined classes as well as imported classes. This way, we can distinguish programming language classes from mock classes and from the external classes that represent the component under test.

Table 5.5 shows most of the typing rules for the specification language extended with programming language classes. Rule T-Spec ensures that the classes of the component under test are included in the type context Δ and that the imported programming language classes are included in the type context Π. Furthermore, the class definitions appearing in the specification are type-checked, where the assumed local type context is extended by the global variables and where the type context for programming language classes is enriched by the defined classes themselves. Finally, regarding the same type context, also the main specification statement is checked, which again yields its control context γ.

Rule T-Class equals its pendant of the programming language apart from the necessary adaption of the judgments regarding the the new type context Π. Likewise, the rules T-Con and T-MDef resemble the corresponding programming language rules. Except for the extended assumption context, however, they additionally restrict the body statement of the method or constructor definition to statements that are well-typed in an internal control context. That is, it ensures that the statement is also a syntactical valid statement of the programming language.

As for the statements, on the one hand we introduce new rules dealing with field updates (T-VUpd$_{\mathsf{PCl}}$), method calls (T-Call$_{\mathsf{PCl}}$), and the new class instantiation statement (T-New$_{\mathsf{PCl}}$). On the other hand, the remaining typing rules regarding other statement, are borrowed from the original specification language. The new rules are again almost identical to the corresponding rules of the programming language's type system. Only the control context is added, putting the new statements in internal control context. Besides that, the type context is extended by Π, which is used to verify the correct types of a constructor or, respectively, method call's parameters. In case of a method call, it is also consulted regarding the return type. Since mock classes and classes of the component under test do not provide access to their fields, it is ensured that the three new statements indeed can only be addressed at programming language classes and their instances.

As for the remaining inherited typing rules regarding statements, they are again extended by the new name context. Some of them are also adapted regarding the control context. A block statement, for instance, may appear within a method body but also within a specification statement. The control context of a block statement's body determines also the control context of the whole block

$$[\text{T-Spec}]\frac{\Delta \vdash \overline{cutdecl} : \mathsf{ok} \quad \Pi \vdash \overline{impdecl} : \mathsf{ok} \quad \Pi' = \Pi, cltype(\overline{cldef}) \quad \Gamma' = \Gamma, \overline{x}{:}\overline{T} \quad \Theta = cltype(\overline{mokdecl}) \quad \Gamma'; \Delta; \Pi'; \Theta \vdash \overline{cldef} : \mathsf{ok} \quad \Gamma'; \Delta; \Pi'; \Theta \vdash stmt : \mathsf{ok}^{\gamma}}{\Gamma; \Delta; \Pi' \vdash \overline{cutdecl}\ \overline{impdecl}\ \overline{mokdecl}\ \overline{T}\ \overline{x};\ \overline{cldef}\ \{stmt\} : \Theta^{\gamma}}$$

$$[\text{T-Class}]\frac{\Gamma' = \Gamma, \overline{f}{:}\overline{T}, \texttt{this}{:}C \quad \Gamma'; \Delta; \Pi; \Theta \vdash con : \mathsf{ok} \quad \Gamma'; \Delta; \Pi; \Theta \vdash \overline{mdef} : \mathsf{ok}}{\Gamma; \Delta; \Pi; \Theta \vdash \texttt{class}\ C\{\overline{T}\ \overline{f};\ con\ \overline{mdef}\} : \mathsf{ok}}$$

$$[\text{T-Con}]\frac{\Gamma' = \Gamma, \overline{x}{:}\overline{T}, \overline{x'}{:}\overline{T'} \qquad \Gamma'; \Delta; \Pi; \Theta \vdash stmt : \mathsf{ok}^{int}}{\Gamma; \Delta; \Pi; \Theta \vdash C(\overline{T}\ \overline{x})\{\overline{T'}\ \overline{x'};\ stmt;\ \texttt{return}\} : \mathsf{ok}}$$

$$[\text{T-MDef}]\frac{\Gamma' = \Gamma, \overline{x}{:}\overline{T}, \overline{x'}{:}\overline{T'} \qquad \Gamma'; \Delta; \Pi; \Theta \vdash stmt : \mathsf{ok}^{int} \qquad \Gamma'; \Delta, \Pi, \Theta \vdash e{:}T}{\Gamma; \Delta; \Pi; \Theta \vdash T\ m(\overline{T}\ \overline{x})\{\overline{T'}\ \overline{x'};\ stmt;\ \texttt{return}\ e\} : \mathsf{ok}}$$

$$[\text{T-VUpd}]\frac{\Gamma; \Delta, \Pi, \Theta \vdash e : \Gamma(x)}{\Gamma; \Delta; \Pi; \Theta \vdash x = e : \mathsf{ok}^{int}}$$

$$[\text{T-Block}]\frac{\gamma \in \{act, int\} \quad \Gamma, \overline{x}{:}\overline{T}; \Delta; \Pi; \Theta \vdash stmt : \mathsf{ok}^{\gamma}}{\Gamma; \Delta; \Pi; \Theta \vdash \{\overline{T}\ \overline{x};\ stmt\} : \mathsf{ok}^{\gamma}}$$

$$[\text{T-NewInt}]\frac{C \in dom(\Theta) \quad \Gamma(x) = C}{\Gamma; \Delta; \Pi; \Theta \vdash x = \texttt{new}\ C() : \mathsf{ok}^{int}}$$

$$[\text{T-New}_{\mathsf{PCl}}]\frac{\Gamma(x) = C \quad \Gamma; \Delta, \Pi, \Theta \vdash \overline{e} : \Pi(C)(C).dom}{\Gamma; \Delta; \Pi; \Theta \vdash x = \texttt{new}\ C(\overline{e}) : \mathsf{ok}^{int}}$$

$$[\text{T-Call}_{\mathsf{PCl}}]\frac{\Gamma; \Delta, \Pi, \Theta \vdash e : C \quad \Gamma(x) = C \quad \Gamma; \Delta, \Pi, \Theta \vdash \overline{e} : \Pi(C)(m).dom}{\Gamma; \Delta; \Pi; \Theta \vdash x = e.m(\overline{e}) : \mathsf{ok}^{int}}$$

$$[\text{T-FUpd}_{\mathsf{PCl}}]\frac{\Gamma; \Delta, \Pi, \Theta \vdash e : \Gamma(f)}{\Gamma; \Delta; \Theta \vdash f = e : \mathsf{ok}^{int}}$$

$$[\text{T-Seq}]\frac{\Gamma; \Delta; \Pi; \Theta \vdash stmt_1 : \mathsf{ok}^{\gamma} \quad \Gamma; \Delta; \Pi; \Theta \vdash stmt_2 : \mathsf{ok}^{\gamma}}{\Gamma; \Delta; \Pi; \Theta \vdash stmt_1;\ stmt_2 : \mathsf{ok}^{\gamma}}$$

$$[\text{T-CtrlSub}]\frac{\Gamma; \Delta; \Pi; \Theta \vdash stmt : \mathsf{ok}^{int}}{\Gamma; \Delta; \Pi; \Theta \vdash stmt : \mathsf{ok}^{act}}$$

Table 5.5: Extension by programming classes: type system (stmts)

statement, which thus can be now an active or an internal control context. The instantiation of a mock class as well as incoming call and outgoing call statements are still considered as passive or, respectively, active statements which are only well-typed if the corresponding callee can be found in the commitment context Θ or the assumed test component context Δ, respectively. This way, it is assured

that a programming language class may not occur as a callee within a specification statement. Sequential composition, while-loops, and conditional statements are well-typed within any control context. However, as in the block statement case, the control context of each of these statements is determined by the control context of their sub-constituents. In particular, the conditional statement and the sequential statement are only well-typed if their sub-statements share the same control context. For the sake of brevity, we have omitted some of the rules which are transfered from the original specification language with only minor adaption as mentioned above.

Finally, we need a subsumption rule, T-CtrlSub, regarding the active and internal control context, as each internal statement, i.e., a statement which is also part of the programming language, may also occur in an active specification statement.

The grammar and the type system ensure that class definitions appearing within a well-formed specification also represent well-formed class definitions regarding the programming language. This eases the extension of the operational semantics. For, we can borrow the internal rules for internal method and constructor calls as well as for field updates from the operational semantics of the programming language without the need for any modifications. Since the extension only concerns internal computations, we do not have to extend the external transition rules.

Also the extension of the code generation is straightforward: the class definitions can be transfered to the test program without any adaption. Moreover, the newly introduced statements may only occur in active control context. In particular, they may only occur within an incoming call statement such that the code generation algorithm will let them become part of the corresponding method or constructor body. Hence, the statements can be copied into the corresponding method or constructor definition of the resulting test program.

5.3 Subtyping and inheritance

Two important concepts of object-oriented programming languages are *inheritance* and *subtyping*. The concept of inheritance facilitates the re-use of code. In the context of class-based object-oriented languages, code re-use operates on classes, i.e., one class can *inherit* the field and method definitions of another class. Subtyping refers to the concept where types are put into a partial order relation giving rise to *type compatibility*. More specifically, within a program, an expression of a certain type can be replaced by an expression of a smaller type without compromising well-typedness. Although inheritance and subtyping actually represent two different concepts, most of the mainstream class-based programming languages merge them to one concept that we will refer to as *subclassing*: A class that inherits the code from another class represents a smaller, i.e. a *subclass*, of the code-donating *superclass*. Integrating subtyping and inheritance is possible due to the fact that classes represent types.

p	$::= \overline{impldecl};\ \overline{T}\,\overline{x};\ \overline{cldef}\ \{stmt;\ \texttt{return}\}$	program
$impdecl$	$::= \texttt{import}\ C$	import
$cldef$	$::= \texttt{class}\ C\ \texttt{extends}\ C\{\overline{T}\,\overline{f};\ con\ \overline{mdef}\}$	class definition
con	$::= C(\overline{T}\,\overline{x})\{\overline{T}\,\overline{x};\ stmt;\ \texttt{return}\}$	constructor
$mdef$	$::= T\ m(\overline{T}\,\overline{x})\{\overline{T}\,\overline{x};\ stmt;\ \texttt{return}\ e\}$	method definition
$stmt$	$::= x = e \mid x = e.m(e,\ldots,e) \mid x = \texttt{new}\ C(e,\ldots,e)$	statements
	$\mid f = e \mid \varepsilon \mid stmt; stmt \mid \{\overline{T}\,\overline{x};\ stmt\}$	
	$\mid \texttt{while}\ (e)\ \{stmt\} \mid \texttt{if}\ (e)\ \{stmt\}\ \texttt{else}\ \{stmt\}$	
	$\mid x = \texttt{super}.m(\overline{e}) \mid \texttt{super}(\overline{e})$	
e	$::= x \mid f \mid \texttt{null} \mid \texttt{this} \mid \texttt{op}(e,\ldots,e)$	expressions

Table 5.6: *Japl* with subclassing: syntax

We want to investigate the impact of introducing subclassing into our testing approach. To this end, we extend our *programming language* with single-inheritance and single-subtyping by introducing the notion of subclassing. That is, a class may have at most one superclass. In particular, we do not account for additional concepts that would introduce polymorphism, like, for instance, *Java*'s notion of interfaces. Subclasses may provide new implementations of inherited methods. In one word, we want to allow for *overriding*. To keep the extension simple, however, we restrict overriding, such that the signature of the new method definition entails exactly the same parameter and return types. In particular we do not allow covariance on return types and we do not deal with overwriting either. However, within a redefining method body we provide a keyword `super` which allows to execute the implementation of the *inherited* method definition. Finally, the extension of our language shall implement dynamic method dispatch, meaning that the method body to be executed due to a method invocation is determined not statically but at runtime.

The syntactical extension of the programming language is shown in Table 5.6. Class definitions include a reference to the superclass. Furthermore, the set of statements is extended by a method call statement and a constructor call statement addressing the method implementation of the superclass. We assume the existence of a class `Object` which provides no fields, no method definitions, and only an empty constructor body. Thus, all classes imported or defined within the program have at least class `Object` as superclass.

As for the type system, we have to incorporate the subtyping relation. To this end, we extend the type of a class with the class name of its superclass:

$$T ::= clnames \times ((MNames \cup clnames) \rightharpoonup (U \times \ldots \times U \rightarrow U))$$

Likewise, we have to adapt the auxiliary function *cltype*. Recall that *cltype* is used to extract the type of a class from its definition. We modify the original

definition, given in Section 2.2, in that the type of a class shall also include the typing information regarding its inherited methods. Thus, the function *cltype* needs to consult the typing context in order to find out the method types of the superclass:

$$\begin{array}{l} \mathit{cltype}(\ \Delta,\ \texttt{class}\ C\ \texttt{extends}\ D\{\ \overline{T}\,\overline{f};\ \mathit{con}\ \overline{\mathit{mdef}}\ \}\ ) \stackrel{\text{def}}{=} C{:}(D,I)\ \text{, where} \\ \quad I : (\mathit{MNames} \cup \mathit{CNames}) \rightharpoonup (U \times \ldots \times U \to U); \\ \quad n \mapsto \begin{cases} (\overline{T} \to C) & \text{if } n = C \text{ and } C(\overline{T}\,\overline{x})\{\overline{T'}\,\overline{x'};\ \mathit{stmt};\ \texttt{return}\} \in \overline{\mathit{mdef}} \\ (\overline{T} \to T) & \text{if } n = m \text{ and } T\ m(\overline{T}\,\overline{x})\{\overline{T'}\,\overline{x'};\ \mathit{stmt};\ \texttt{return}\ e\} \in \overline{\mathit{mdef}} \\ (\overline{T} \to T) & \text{if } \Delta(D) = (E, I') \text{ and } I'(n) = (\overline{T}, T) \end{cases} \end{array}$$

We show in Table 5.7 new and, respectively, modified typing rules according to the typing rules of Table 2.2 and Table 2.9. As mentioned above, the domain of the auxiliary function *cltype* has been adapted, such that also rule T-Prog has to be changed correspondingly. Note, that a class definition might use a class as its superclass whose definition was given ahead within the program code. Thus, the commitment context is determined incrementally. Specifically, we assume that $\overline{\mathit{cldef}}$ consists of the sequence $\mathit{cldef}_1\ \mathit{cldef}_2\ \ldots\ \mathit{cldef}_n$.

The rule T-Class is merely modified in that we adapted the class definition code in the conclusion judgment. In particular we didn't change the handling of the fields. It is a crucial point that still only the fields of the defining class are incorporated into the local type context. For, the consequence is that the constructor and the method bodies do not have access to fields provided by the superclass. This way we stipulate a field access policy where all fields are considered as private in the sense that they can be accessed by instances of the defining class, only. We will see later that this decision influences the observability of some interaction.

The rules T-SupCall and T-SupNew implement the type check for the new statements, namely for the call statements that address inherited method or constructor code. Since a class type also incorporates the type information of inherited methods, we do not need to descent the type succession but we can check well-typedness directly by consulting the type of the class that contains the `super` call. To determine the class in question we only have to lookup the type of `this`. All other premises are equal to the corresponding premises of the typing rules regarding the conventional method and constructor calls. This entails a slight abuse of notation, since now the type of a class does not only consist of the method and constructor type function but it is now a pair consisting of the class name of the super class and the mentioned type function. However, we keep the notation, that is, although the type of a class C is now of the form $\Delta(C) = (D, I)$, we still write

$$\Delta(C)(m).\mathit{dom} \quad \text{and} \quad \Delta(C)(m).\mathit{ran}$$

to denote the domain and, respectively, the range of the type function I. Similarly, we write

$$\Delta(C).\mathit{supcl}$$

$$[\text{T-Prog}]\ \frac{\begin{array}{c}\Gamma' = \Gamma, \overline{x}{:}\overline{T} \\ \Theta_1 = \mathit{cltype}(\Delta, \mathit{cldef}_1)\ \ldots\ \Theta_n = \mathit{cltype}(\Delta, \Theta_{n-1}, \mathit{cldef}_n) \\ \Gamma; \Delta \vdash \overline{\mathit{impdecl}} : \mathsf{ok} \quad \Gamma'; \Delta, \Theta_n \vdash \overline{\mathit{cldef}} : \mathsf{ok} \quad \Gamma'; \Delta, \Theta_n \vdash \mathit{stmt} : \mathsf{ok}\end{array}}{\Gamma; \Delta \vdash \overline{\mathit{impdecl}};\ \overline{T}\ \overline{x};\ \mathit{cldef}_1 \ldots \mathit{cldef}_n\ \{\mathit{stmt};\ \mathtt{return}\} : \Theta_n}$$

$$[\text{T-Class}]\ \frac{\Gamma' = \Gamma, \overline{f}{:}\overline{T}, \mathtt{this}{:}C \qquad \Gamma'; \Delta \vdash \mathit{con} : \mathsf{ok} \qquad \Gamma'; \Delta \vdash \overline{\mathit{mdef}} : \mathsf{ok}}{\Gamma; \Delta \vdash \mathtt{class}\ C\ \mathtt{extends}\ D\{\overline{T}\ \overline{f};\ \mathit{con}\ \overline{\mathit{mdef}}\} : \mathsf{ok}}$$

$$[\text{T-SupCall}]\ \frac{\Gamma(\mathtt{this}) = C \qquad \Gamma(x) = \Delta(C)(m).\mathit{ran} \qquad \Gamma; \Delta \vdash \overline{e} : \Delta(C)(m).\mathit{dom}}{\Gamma; \Delta \vdash x = \mathtt{super}.m(\overline{e}) : \mathsf{ok}}$$

$$[\text{T-SupNew}]\ \frac{\Gamma(\mathtt{this}) = C \qquad \Gamma; \Delta \vdash \overline{e} : \Delta(C)(C).\mathit{dom}}{\Gamma; \Delta \vdash \mathtt{super}(\overline{e}) : \mathsf{ok}}$$

Table 5.7: *Japl* with subclassing: type system (stmts)

to denote the superclass D of C.

As mentioned earlier, an instance of a class has only access to the fields that are defined in that class. This is a central aspect for the operational semantics. First of all, however, we should have a look at a small example which will reveal that the above statement is actually not completely true, if sub-classing comes into play. Consider a code fragment which consists of the definition of two classes C and D.

```
1  C extends object {
2    T x;
3    T meth1() { x = ... }
4  }
5
6  D extends C {
7    T y;
8    T meth2() { y = ...; z = super.meth1(); ... }
9  }
```

Each class definition consists of a variable declaration and a method definition. Furthermore, class D is a subclass of class C. Now assume that we have an instance o of class D and we call its method *meth2*. According to the method body of *meth2*, its execution will change the value of the variable y. This is fine, as the variable y is declared within class D. Moreover, it is true that the method body of *meth2* must not access the variable x, as it is *not* declared within the definition of D. However, *meth2* may call the method *meth1* inherited from class C. Method *meth1*, in turn, may access and even change the variable x. Therefore, although object o is an instance of class D, it may access the inherited variable x – but only by means of an invocation of an inherited method.

The consequence regarding the operational semantics is that object o is represented twice in the heap h of the program: one entry in h stores the value of x, the other entry the value of y. The idea is that one entry represents o as an object of class C and the other entry represents o as an instance of class D. Regarding the execution of method *meth2*, o can be considered as an object of class D, hence it may access the corresponding entry in h, only. Similarly, during the execution of *meth1* we may access o via the other entry only.

On account of this, we have to change the heap function in that it does not only map object names o to objects (C, F) consisting of the object's class C and the object's field values F (cf. Definition 2.3.1), but the domain of the heap is extended by class names. Thus, a heap maps pairs of object and class names to objects. That is, the set of heap functions is redefined to:

$$\mathsf{H} \stackrel{\text{def}}{=} (\mathit{CNames} \times N) \rightharpoonup \mathit{Obj}.$$

Note, that an object still is represented by a pair (C, F) consisting of the object's field function F but also of its class C. The class C is the class from which the object has been instantiated. For instance, regarding the above example object o of class D has two representations in the heap h. In particular, it is

$$h(D, o) = (D, \{y \mapsto v_x\}) \quad \text{and} \quad h(C, o) = (D, \{x \mapsto v_y\}).$$

Moreover, we introduce a new auxiliary variable `cls` which is used to determine the class of the currently executed method body. Specifically, assuming a heap h, a global variable function v, and a local variable function list μ it is

$$C = [\![\texttt{cls}]\!]_h^{\mathsf{v},\mu}$$

the class that implements the currently executed method body. Therefore, we can access the currently executed object as it is presented to the currently executed method by means of the expression $h([\![\texttt{cls}]\!]_h^{\mathsf{v},\mu}, [\![\texttt{this}]\!]_h^{\mathsf{v},\mu})$.

Apart from the field access mechanism, the above example additionally demonstrated the invocation of the inherited methods *meth1* by using the keyword `super`. Since the class type provides the name of its superclass, extending the operational semantics with the `super` calls is straightforward: instead of looking up and expanding the method body of the executing class we use the method body that is provided by the superclass.

In the example, we actually didn't need to explicitly choose the inherited method implementation by calling $\texttt{super}.\mathit{meth}1()$ but, since D does not override *meth1*, we could have called $\texttt{this}.\mathit{meth}1()$, as well, getting the same result. Hence, we assume a *dynamic dispatching* of method invocations. This dispatching mechanism is realized as follows. Within a sub-class D, for each inherited method

$$T\ m(\overline{T}\ \overline{x})$$

which is not replaced by new code in terms of a new method definition, we assume an invisible method definition as follows:

$$T\ m(\overline{T}\ \overline{x})\{\ x = \texttt{super}.m(\overline{x});\ \texttt{return}(x)\ \}.$$

As for the above example, for instance, we assume a hidden method definition of the form

$$T\ meth1()\ \{\ T\ x;\ x = super.meth1();\ return(x)\ \},$$

extending the explicitly given class definition of class D. Thus, in general, the execution of a method $\texttt{this}.m(\overline{e})$ does not require a complicated look-up mechanism in order to find the class that actually implements the method. Instead, the corresponding implementation is always provided by the calling class which then might possibly call the inherited method explicitly by means of a `super` call – if it didn't override it by real user code.

Having discussed the underlying modifications of the operational semantics, let us have a look at the corresponding rules. Regarding the internal steps, we only have to change the rules of Table 2.7 that deal with field updates, internal method calls and internal object creation. Moreover, we have to add new rules for calling methods or constructors of the superclass. The rules are given in Table 5.8.

$$[\textsc{FUpd}]\ \frac{\begin{array}{c} o = [\![\texttt{this}]\!]_h^{\mathsf{v},\mu} \quad C = [\![\texttt{cls}]\!]_h^{\mathsf{v},\mu} \quad (C', \mathsf{F}) = h(C, o) \\ h' = h[(C, o) \mapsto (C', \mathsf{F}[f \mapsto [\![e]\!]_h^{\mathsf{v},\mu}])] \end{array}}{(h, \mathsf{v}, (\mu, f = e; mc) \circ \mathsf{CS}^b) \rightsquigarrow (h', \mathsf{v}, (\mu, mc) \circ \mathsf{CS}^b)}$$

$$[\textsc{Call}]\ \frac{\begin{array}{c} o = [\![e]\!]_h^{\mathsf{v},\mu} \quad C = h(_, o).class \quad \overline{T}\,\overline{x} = mparams(C, m) \quad \overline{T_l}\,\overline{x_l} = mvars(C, m) \\ \Delta \not\vdash C : [\![\ldots]\!] \quad \mathsf{v}_l = \{\texttt{cls} \mapsto C, \texttt{this} \mapsto o, \overline{x} \mapsto [\![\overline{e}]\!]_h^{\mathsf{v},\mu}, \overline{x_l} \mapsto ival(\overline{T_l})\} \end{array}}{(h, \mathsf{v}, (\mu, x = e.m(\overline{e}); mc) \circ \mathsf{CS}^b) \rightsquigarrow (h, \mathsf{v}, (\mathsf{v}_l, mbody(C, m)) \circ (\mu, \texttt{rcv}\ x;\ mc) \circ \mathsf{CS}^b)}$$

$$[\textsc{New}]\ \frac{\begin{array}{c} o \in N \setminus dom(h) \quad h' = h[(C, o) \mapsto Obj_{\perp}^{C}] \quad \overline{T}\,\overline{x} = cparams(C) \quad \overline{T_l}\,\overline{x_l} = cvars(C) \\ \mathsf{v}_l = \{\texttt{cls} \mapsto C, \texttt{this} \mapsto o, \overline{x} \mapsto [\![\overline{e}]\!]_h^{\mathsf{v},\mu}, \overline{x_l} \mapsto ival(\overline{T_l})\} \end{array}}{\begin{array}{c} (h, \mathsf{v}, (\mu, x = \texttt{new}\ C(\overline{e}); mc) \circ \mathsf{CS}^b) \rightsquigarrow \\ (h', \mathsf{v}, (\mathsf{v}_l, cbody(C);\ \texttt{return this}) \circ (\mu, \texttt{rcv}\ x;\ mc) \circ \mathsf{CS}^b) \end{array}}$$

$$[\textsc{SupCall}]\ \frac{\begin{array}{c} o = [\![e]\!]_h^{\mathsf{v},\mu} \quad C = [\![\texttt{cls}]\!]_h^{\mathsf{v},\mu} \quad C' = \Theta(C).supcl \\ \overline{T}\,\overline{x} = mparams(C, m) \quad \overline{T_l}\,\overline{x_l} = mvars(C, m) \\ \Delta \not\vdash C' : [\![\ldots]\!] \quad \mathsf{v}_l = \{\texttt{cls} \mapsto C, \texttt{this} \mapsto o, \overline{x} \mapsto [\![\overline{e}]\!]_h^{\mathsf{v},\mu}, \overline{x_l} \mapsto ival(\overline{T_l})\} \end{array}}{\begin{array}{c} (h, \mathsf{v}, (\mu, x = \texttt{super}.m(\overline{e}); mc) \circ \mathsf{CS}^b) \rightsquigarrow \\ (h, \mathsf{v}, (\mathsf{v}_l, mbody(C, m)) \circ (\mu, \texttt{rcv}\ x;\ mc) \circ \mathsf{CS}^b) \end{array}}$$

$$[\textsc{SupNew}]\ \frac{\begin{array}{c} o = [\![e]\!]_h^{\mathsf{v},\mu} \quad C = [\![\texttt{cls}]\!]_h^{\mathsf{v},\mu} \quad C' = \Theta(C).supcl \\ \overline{T}\,\overline{x} = mparams(C, m) \quad \overline{T_l}\,\overline{x_l} = mvars(C, m) \\ \Delta \not\vdash C' : [\![\ldots]\!] \quad \mathsf{v}_l = \{\texttt{cls} \mapsto C, \texttt{this} \mapsto o, \overline{x} \mapsto [\![\overline{e}]\!]_h^{\mathsf{v},\mu}, \overline{x_l} \mapsto ival(\overline{T_l})\} \end{array}}{\begin{array}{c} (h, \mathsf{v}, (\mu, \texttt{super}(\overline{e}); mc) \circ \mathsf{CS}^b) \rightsquigarrow \\ (h, \mathsf{v}, (\mathsf{v}_l, cbody(C, m);\ \texttt{return this}) \circ (\mu, \texttt{rcv}\ _;\ mc) \circ \mathsf{CS}^b) \end{array}}$$

Table 5.8: *Japl* with subclassing: operational semantics (int.)

Rule FUpd is modified in that we explicitly have to look up the class C whose method body is currently in execution. For, as mentioned above, the class and

program:	environment:
C2 `extends` *C1* {	*C1* `extends` *Object* {
...	...
}	}
C4 `extends` *C3* {	*C3* `extends` *C2* {
...	...
}	}

Table 5.9: Example: cross-border inheritance

the object name are needed to get the corresponding object representation (C', F) from the heap. Finally, as in the original rule, first the field function F and then, correspondingly, the heap is updated.

Regarding an internal method call, we have to find out the class from which the callee object o has been instantiated. To this end, we consult the heap regarding o. Note that for all entries of o in the heap, the yielded class is the same. The rest of the rule is quite similar to the original rule. However, the new local variable function v_l additionally stores the class C in `cls`, as its method is about to be executed. Moreover, we have to consult the assumption context Δ in order to check that C is indeed not an external class.

The modification regarding Rule New are similar to the modifications of Rule Call. Though, we do not have to look up the type C.

In order to call an inherited method via the keyword `super`, we first have to find out the caller class C via the variable `cls`. This is shown in Rule SupCall. Afterwards, we can look up C's superclass C' and, if C' is also a program class, then we can execute its method implementation as it has been explained for Rule Call, already.

Again, similar to the Rule SupCall, Rule SupNew finds out the superclass of the caller class and executes its constructor, if the superclass is a program class.

It may happen that the program extends a class of the environment or vice versa meaning that sub-class and super-class are defined on different sides. This has the effect that calls of inherited methods or constructors may cross the interface. To understand the consequences, consider the example given in Table 5.9. In the example, some environment class $C1$ is extended by a program class $C2$. Class $C2$ is again extended by an environment class $C3$ which in turn is extended by a program class $C4$. Now let us assume that an instance o of $C4$ calls an inherited method $m3$ of $C3$. This results in a cross-border method call, where the environment executes the method body of $m3$ provided by $C3$. In order to

$$[\textsc{CallO}]\ \frac{a = \nu(\Theta').\langle \mathit{call}\ C.o.m(\overline{v})\rangle! \qquad \Delta \vdash o : C}{\begin{array}{c}\Delta \vdash (h, \mathsf{v}, (\mu, x = e.m(\overline{e}); mc) \circ \mathsf{CS}^b) : \Theta \xrightarrow{a} \\ \Delta \vdash (h, \mathsf{v}, (\mu, \mathtt{rcv}\ x{:}T;\ mc) \circ \mathsf{CS}^b) : \Theta, \Theta'\end{array}}$$

where $o = [\![e]\!]_h^{\mathsf{v},\mu}$, $\overline{v} = [\![\overline{e}]\!]_h^{\mathsf{v},\mu}$, $T = \Delta^2(o)(m).ran$, and $\Theta' = \mathsf{new}(h, \overline{v}, \Theta)$

$$[\textsc{CallI}]\ \frac{a = \nu(\Delta').\langle \mathit{call}\ C.o.m(\overline{v})\rangle? \qquad \Delta \vdash a : \Theta}{\begin{array}{c}\Delta \vdash (h, \mathsf{v}, \mathsf{CS}^{eb}) : \Theta \xrightarrow{a} \\ \Delta, \Delta' \vdash (h, \mathsf{v}, (\mathsf{v}_l, \mathit{mbody}(C, m)) \circ \mathsf{CS}^{eb}) : \Theta\end{array}}$$

where $C = \Theta(o)$, $\overline{T}\,\overline{x} = \mathit{mparams}(C, m)$, $\overline{T'}\,\overline{x'} = \mathit{mvars}(C, m)$, and $\mathsf{v}_l = \{\mathtt{cls} \mapsto C, \mathtt{this} \mapsto o, \overline{x} \mapsto \overline{v}, \overline{x'} \mapsto \mathit{ival}(\overline{T'})\}$

Table 5.10: *Japl* with subclassing: operational semantics (ext.)

potentially access fields of $C3$, object o is considered as an object of $C3$ during the execution of $m3$ as we have explained above. However, $m3$ may itself call an inherited method $m2$ of $C2$ which, in turn, may call an inherited method $m1$ of $C1$. Again, object o has to be considered as an object of $C1$ in order to access fields of $C1$. Summarizing, object o shows up twice as callee object in the environment – however, the first call needed to consider o as an object of $C3$ and the second call casted o to an object of $C1$. Therefore, regarding the external steps, we have to equip the communication labels a for method calls with a class type C of the callee object o, such that $a = \nu(\Theta').\langle \mathit{call}\ C.o.m(\overline{v})\rangle!$ and, respectively, $a = \nu(\Delta').\langle \mathit{call}\ C.o.m(\overline{v})\rangle?$. Apart from that, we only have to implement minor adaptions regarding the rules for incoming and outgoing method calls of the external semantics. The rules are given in Table 5.10.

Part II

Testing Multi-threaded Components

In the previous part of this thesis we presented a formal framework for testing object-oriented components in a *sequential* setting. That is, the language allowed for a single-threaded flow of control, only. In the following part, we suggest an extension of the framework regarding multi-threaded components. In particular, we will extend the underlying programming language with the notion of *threads*. In languages like *Java* and C$^\sharp$ objects are passive entities residing in the heap of the program – instantiated from classes that serve as "generators of state"; the active part of the program is represented by threads. Indeed, in a multi-threaded setting, there is also a mechanism for *"generating new activity"*, i.e., for creating new threads. Thus, we extend our previous work by thread instantiation from *thread classes*, meaning that new activities can be dynamically spawned from "templates".

Correspondingly, we have to adapt the test specification language. The underlying idea is that we cope with multi-threading by providing a specification statement for each thread. Hence, only the order of interactions which belong to the same thread is specified.

Finally, we sketch how the code generation algorithm of the single-threaded setting can be modified in order to generate test programs also for multi-threaded components.

Chapter 6

Concurrent programming language – *CoJapl*

6.1 Syntax

As mentioned in the introduction, we incorporate concurrency into the programming language *Japl* of Chapter 2 by means of *thread classes*. The corresponding syntactical modifications are rather straightforward. The grammar for the resulting concurrent *Java*-like language, *CoJapl*, is given in Table 6.1. Once again, to emphasize the extending character, we grayed out the constructs that are inherited from the sequential programming language *Japl*. The grammar shows that a thread class definition resembles the definition of an object class constructor. That is, we do not embark on the strategy of *Java* or C♯, where thread classes are realized by means of designated object classes. Instead, introducing a new kind of classes allows for a clear separation of concerns, as object classes are the generators of *state* while thread classes are used to generate *activity*.

The signature of a thread class provides a thread class name C and a list of formal parameters. The body of a thread class consists of a body statement *stmt* and a concluding `return`. Thus, a *CoJapl* program p does not only provide a sequence of class definitions $\overline{cldef}$ but additional it allows to define a sequence of thread class definitions $\overline{tdef}$.

The counterpart of a thread class definition is the `spawn` statement, which is used to create a new thread instance from a thread class and which, thus, has some similarities with the `new` statement. It specifies the name of the thread class which serves as the code template for the new thread. A sequence of expressions $\overline{e}$ represent the actual parameter of the new thread. The `spawn` statement is an assignment. It allows to store the *thread identifier* of the new thread in a variable x. A thread identifier is comparable with an object name insofar as it uniquely identifies a thread. Note however, that a thread is not allocated on the heap. Specifically, we assume the existence of another infinite set `thread` which serves

p	$::= \overline{impdecl};\ \overline{T}\,\overline{x};\ \overline{cldef}\ \overline{tdef}\ \{\ stmt;\ \texttt{return}\ \}$	program
$impdecl$	$::= \texttt{import}\ C$	import declaration
$cldef$	$::= \texttt{class}\ C\{\ \overline{T}\,\overline{f};\ con\ \overline{mdef}\ \}$	class definition
con	$::= C(\overline{T}\,\overline{x})\{\ \overline{T}\,\overline{x};\ stmt;\ \texttt{return}\ \}$	constructor
$mdef$	$::= T\ m(\overline{T}\,\overline{x})\{\ \overline{T}\,\overline{x};\ stmt;\ \texttt{return}\ e\ \}$	meth. definition
$tdef$	$::= \texttt{thread}\ C(\overline{T}\,\overline{x})\{\ stmt;\ \texttt{return}\ \}$	thread class definition
$stmt$	$::= x = e \mid x = e.m(e,\ldots,e) \mid x = \texttt{new}\ C(e,\ldots,e)$	statements
	$\mid f = e \mid \varepsilon \mid stmt; stmt \mid \{\overline{T}\,\overline{x};\ stmt\}$	
	$\mid \texttt{while}\ (e)\ \{stmt\} \mid \texttt{if}\ (e)\ \{stmt\}\,\texttt{else}\ \{stmt\}$	
	$\mid x = \texttt{spawn}\ C(e,\ldots,e)$	
e	$::= x \mid f \mid \texttt{null} \mid \texttt{this} \mid \texttt{op}(e,\ldots,e) \mid \texttt{tid} \mid \texttt{tclass}$	expressions

Table 6.1: *CoJapl* language: syntax

as the domain of thread identifiers. For the sake of simplicity we do not allow to pass around thread identifiers in terms of a parameter or a return value. Within the thread itself, its thread identifier can be found out by means of the new expression `tid`. Moreover, a thread may also identify the name of its thread class using the expression `tclass`.

6.2 Static semantics

Similar to the syntax definition, also the type system needs only small changes regarding the concurrency extension. Concerning the typing rules for the syntactical constituents up to statements, given in Table 6.2, we only have to modify Rule T-Prog and to add two rules for the two new constructs, namely for thread definitions and for the `spawn` statement. Apart from these changes, we keep the rules from Table 2.2 and, respectively, Table 2.9 without any changes.

As for the Rule T-Prog, we only have change the definition of the commitment context Θ, as not only the object classes but also the thread classes are provided to the program's environment. This is essential as it enables an external component to instantiate a thread class defined in the program. On account of this, we have to extend the definition of the auxiliary function *cltype*. While in the sequential setting the function *cltype* was used in order to extract the typing information of a class from the corresponding object class definition, in the concurrent setting it additionally has to extract the typing information from thread class definitions. Thus, we extend the definition given in Section 2.2 by the following definition:

$$cltype(\ \texttt{thread}\ C(\overline{T}\,\overline{x})\{\ stmt;\ \texttt{return}\}\ ) \stackrel{\text{def}}{=} C : \overline{T}$$

Therefore, in contrast to the type of an object class, a thread class type consists of its parameter type list $\overline{T}$, only. Note, specifically, that a thread class type is not

a functional type because a thread does not provide a return value. Furthermore, note in this context that we use the same domain *CNames* for, both, object classes and thread classes. Hence, we assume all names of object classes *and* thread classes to be unique within the program. In particular, a class name C is at most either typed as an object class or as a thread class.

Rule T-TDEF deals with the syntax check of thread class definitions. Again, the rule is almost identical to the corresponding rule for constructors T-CON. The local type context is extended by the formal parameters which is then used to type check the body statement *stmt* of the thread class.

The `spawn` statement is type-checked by means of Rule T-SPAWN. Such a statement is well-typed if the variable x is a thread variable and if the class name C, indeed, refers to a thread class definition such that the thread class's formal parameters and the actual parameters match regarding their types.

Table 6.3 deals with the typing rules for expressions. According to Table 2.3, we only add two new typing rules and keep the rest unchanged. Both the new expression, `tid` and `tclass`, are well-typed in any type context, as a statement is always executed in context of a specific thread. Specifically, we will see in the next section concerning the operational semantics that also the main body statement of the program will be provided with a thread identifier n_{main} and a designated thread class name *Main*.

6.3 Operational semantics

As mentioned earlier, thread classes serve as generators of activity. Indeed, the required modifications of the operational semantics due to the introduction of thread classes mostly affect the call stack, as it represents the active code. Recall, that in *Japl* the call stack captures the sequential flow of control by means of a list of activation records. That is, in the sequential setting, a call stack is of the form

$$\mathsf{CS} = \mathsf{AR}_0 \circ \mathsf{AR}_1^b \circ \mathsf{AR}_2^b \ldots \circ \mathsf{AR}_n^b,$$

where the activation records AR_1^b to AR_n^b are either externally or internally blocked. Hence, they are of the form

$$\mathsf{AR}^b ::= (\mu, \texttt{rcv}\ x;\ mc) \mid (\mu, \texttt{rcv}\ x{:}T;\ mc).$$

The topmost activation record AR_0, however, is either currently in execution or it is externally blocked, i.e., in the latter case the program waits for an incoming communication from the environment. Summarizing, we can say that the form of the call stack as well as the rules of the operational semantics of *Japl* allow to reduce only the topmost statement of the topmost activation record, if at all. This way, the sequential flow of control is ensured. In particular, the operational semantics adheres to the order of the sequentially composed statements.

Regarding the multi-threaded setting, it is natural to use the above mentioned call stack mechanism for each thread, as each thread on its own shall adhere

$$[\text{T-Prog}]\ \frac{\Gamma' = \Gamma, \overline{x}{:}\overline{T} \quad \Theta = \mathit{cltype}(\overline{\mathit{cldef}}), \mathit{cltype}(\overline{\mathit{tdef}}) \quad \Gamma; \Delta \vdash \overline{\mathit{impdecl}} : \mathsf{ok} \quad \Gamma'; \Delta, \Theta \vdash \overline{\mathit{cldef}} : \mathsf{ok} \quad \Gamma'; \Delta, \Theta \vdash \overline{\mathit{tdef}} : \mathsf{ok} \quad \Gamma'; \Delta, \Theta \vdash \mathit{stmt} : \mathsf{ok}}{\Gamma; \Delta \vdash \overline{\mathit{impdecl}};\ \overline{T}\,\overline{x};\ \overline{\mathit{cldef}}\ \overline{\mathit{tdef}}\ \{\mathit{stmt};\ \texttt{return}\} : \Theta}$$

$$[\text{T-Import}]\ \frac{C \in \mathit{dom}(\Delta)}{\Gamma; \Delta \vdash \texttt{import}\ C : \mathsf{ok}}$$

$$[\text{T-Class}]\ \frac{\Gamma' = \Gamma, \overline{f}{:}\overline{T}, \texttt{this}{:}C \quad \Gamma'; \Delta \vdash \mathit{con} : \mathsf{ok} \quad \Gamma'; \Delta \vdash \overline{\mathit{mdef}} : \mathsf{ok}}{\Gamma; \Delta \vdash \texttt{class}\ C\{\overline{T}\,\overline{f};\ \mathit{con}\ \overline{\mathit{mdef}}\} : \mathsf{ok}}$$

$$[\text{T-Con}]\ \frac{\Gamma' = \Gamma, \overline{x}{:}\overline{T}, \overline{x'}{:}\overline{T'} \quad \Gamma'; \Delta \vdash \mathit{stmt} : \mathsf{ok}}{\Gamma; \Delta \vdash C\ (\overline{T}\,\overline{x})\{\overline{T'}\,\overline{x'};\ \mathit{stmt};\ \texttt{return}\} : \mathsf{ok}}$$

$$[\text{T-MDef}]\ \frac{\Gamma' = \Gamma, \overline{x}{:}\overline{T}, \overline{x'}{:}\overline{T'} \quad \Gamma'; \Delta \vdash \mathit{stmt} : \mathsf{ok} \quad \Gamma'; \Delta \vdash e : T}{\Gamma; \Delta \vdash T\ m(\overline{T}\,\overline{x})\{\overline{T'}\,\overline{x'};\ \mathit{stmt};\ \texttt{return}\ e\} : \mathsf{ok}}$$

$$[\text{T-TClass}]\ \frac{\Gamma' = \Gamma, \overline{x}{:}\overline{T} \quad \Gamma'; \Delta \vdash \mathit{stmt} : \mathsf{ok}}{\Gamma; \Delta \vdash \texttt{thread}\ C(\overline{T}\,\overline{x})\{\mathit{stmt};\ \texttt{return}\} : \mathsf{ok}}$$

$$[\text{T-VUpd}]\ \frac{\Gamma; \Delta \vdash e : \Gamma(x)}{\Gamma; \Delta \vdash x = e : \mathsf{ok}} \qquad [\text{T-FUpd}]\ \frac{\Gamma; \Delta \vdash e : \Gamma(f)}{\Gamma; \Delta \vdash f = e : \mathsf{ok}}$$

$$[\text{T-Call}]\ \frac{\Gamma; \Delta \vdash e : C \quad \Gamma(x) = \Delta(C)(m).\mathit{ran} \quad \Gamma; \Delta \vdash \overline{e} : \Delta(C)(m).\mathit{dom}}{\Gamma; \Delta \vdash x = e.m(\overline{e}) : \mathsf{ok}}$$

$$[\text{T-New}]\ \frac{\Gamma(x) = C \quad \Gamma; \Delta \vdash \overline{e} : \Delta(C)(C).\mathit{dom}}{\Gamma; \Delta \vdash x = \texttt{new}\ C(\overline{e}) : \mathsf{ok}}$$

$$[\text{T-Seq}]\ \frac{\Gamma; \Delta \vdash \mathit{stmt}_1 : \mathsf{ok} \quad \Gamma; \Delta \vdash \mathit{stmt}_2 : \mathsf{ok}}{\Gamma; \Delta \vdash \mathit{stmt}_1;\ \mathit{stmt}_2 : \mathsf{ok}} \qquad [\text{T-Block}]\ \frac{\Gamma, \overline{x}{:}\overline{T}; \Delta \vdash \mathit{stmt} : \mathsf{ok}}{\Gamma; \Delta \vdash \{\overline{T}\,\overline{x};\ \mathit{stmt}\} : \mathsf{ok}}$$

$$[\text{T-While}]\ \frac{\Gamma; \Delta \vdash e : \mathsf{bool} \quad \Gamma; \Delta \vdash \mathit{stmt} : \mathsf{ok}}{\Gamma; \Delta \vdash \texttt{while}\ (e)\ \{\mathit{stmt}\} : \mathsf{ok}}$$

$$[\text{T-Cond}]\ \frac{\Gamma; \Delta \vdash e : \mathsf{bool} \quad \Gamma; \Delta \vdash \mathit{stmt}_1 : \mathsf{ok} \quad \Gamma; \Delta \vdash \mathit{stmt}_2 : \mathsf{ok}}{\Gamma; \Delta \vdash \texttt{if}\ (e)\ \{\mathit{stmt}_1\}\ \texttt{else}\ \{\mathit{stmt}_2\} : \mathsf{ok}}$$

$$[\text{T-Spawn}]\ \frac{\Gamma(x) = \mathsf{thread} \quad \Gamma; \Delta \vdash \overline{e} : \Delta(C)}{\Gamma; \Delta \vdash x = \texttt{spawn}\ C(\overline{e}) : \mathsf{ok}}$$

Table 6.2: *CoJapl* language: type system (stmts)

$$[\text{T-Var}]\frac{\Gamma(x) = T}{\Gamma; \Delta \vdash x : T} \qquad [\text{T-Field}]\frac{\Gamma(f) = T}{\Gamma; \Delta \vdash f : T}$$

$$[\text{T-Null}]\ \Gamma; \Delta \vdash \texttt{null} : C \qquad [\text{T-This}]\frac{\Gamma(\texttt{this}) = C}{\Gamma; \Delta \vdash \texttt{this} : C}$$

$$[\text{T-Op}]\frac{\Gamma; \Delta \vdash \overline{e} : \mathit{dom}(\Delta(\texttt{op})) \quad \mathit{ran}(\Delta(\texttt{op})) = T}{\Gamma; \Delta \vdash \texttt{op}(\overline{e}) : T}$$

$$[\text{T-Tid}]\ \Gamma; \Delta \vdash \texttt{tid} : \texttt{thread} \qquad [\text{T-Tclass}]\ \Gamma; \Delta \vdash \texttt{tclass} : \mathit{CNames}$$

Table 6.3: *CoJapl* language: type system (exprs)

to the order of the statements. Therefore, in the concurrent extension of our programming language, we will make use of a *set of call stacks*. To this end, we will use *thread configuration mappings*. A thread configuration mapping **tc** is a function of the type

$$\mathbf{TC} = \texttt{thread} \rightharpoonup (\mathit{CNames} \times \mathsf{CS})$$

which maps a thread identifier to its call stack, if the program configuration contains a thread with the thread identifier. Otherwise the mapping is undefined for this identifier. In addition to the call stack, however, a call stack mapping provides the thread class name of the thread. We use

$$\mathbf{tc}(n).\mathit{tclass} \quad \text{and} \quad \mathbf{tc}(n).\mathit{cs}$$

in order to refer to the thread class and to the call stack of a thread identifier n, respectively. As for other mappings, we denote the thread configuration mapping that results from modifying **tc** by mapping the thread identifier n to the thread class C and call stack CS with

$$\mathbf{tc}[n \mapsto (C, \mathsf{CS})].$$

Recall, that this means either an extension of the original domain of **tc** by the new element n or an update of **tc** concerning the image of t. In the latter case, we often write

$$\mathbf{tc}[n \mapsto \mathsf{CS}] \quad \text{as a short form for} \quad \mathbf{tc}[n \mapsto (\mathbf{tc}(n).\mathit{tclass}, \mathsf{CS})],$$

as the execution of the thread may change its call stack but not its thread class, anyway.

Therefore, a configuration of the multi-threaded language *CoJapl* only differs from configuration of the sequential language *Japl* in that the call stack is replaced

by a thread configuration mapping. We redefine the set of configurations *Conf* to

$$Conf \stackrel{\text{def}}{=} (\mathsf{H} \times \mathsf{V} \times \mathbf{TC}).$$

In our concurrency model, not all threads are executed in parallel, but the operational semantics implements a scheduler allowing only one thread at a time to exercise an undetermined number of computation steps. That is, the execution of threads is interleaved. Note, that we embark on a *preemptive* concurrency model. We do not provide specific language constructs like *wait* or *notify* by which a thread could influence the actual scheduling. In particular, neither can a thread explicitly give away the control to another thread nor can it claim execution time.

Now let us discuss the rules of the operational semantics that deal with internal computation steps. They are defined in Table 6.4 and Table 6.5. Regarding the internal rules, the transition from *Japl*'s sequential setting to *CoJapl*'s multi-threaded setting basically consists of the above mentioned replacement of the call stack by the thread configuration mapping within the configurations. Hence, within each transition rule, the call stack is replaced by a thread configuration mapping. Each rule *non-deterministically* chooses a thread identifier n of the thread configuration mapping's domain. Then the associated call stack is reduced much like in the corresponding rules of the sequential setting. Finally, the resulting thread configuration replaces the original configuration within the thread configuration mapping. Note, non-deterministically choosing a thread represents a very simple scheduling policy which, specifically, does not guarantee *fairness*. In other words, theoretically it may happen that a specific thread never gets any execution time.

As for Rule Call, the transition from *Japl* to *CoJapl* does not only entail the above mentioned call stack replacement, but additionally we have to provide the method with values for the expressions `tid` and `tclass`. In order to find out the thread class of the thread n, we do not only look up the thread's call stack in the thread configuration mapping **tc** but also its thread class C_T.

Regarding Rule Spawn some more words are in order. We assume that a thread with thread identifier n_1 is about to spawn a new thread of thread class C. To this end, we choose a new thread name n_2 which is not already in use, hence, which is not in the domain of the thread configuration mapping **tc**, already. The new thread identifier n_2 is returned to the call stack of thread n_1 which correspondingly updates the value of variable x by modifying the local variable function list and the global variables. As for the new thread, we create a new call stack CS_2 which consists of a single activation record, only. In particular, the activation record code is represented by the body of thread class C and its local variable function list consists of the variable function v_l only, capturing the thread's identifier, its class name, and the parameters $\overline{x}$ of the `spawn` statement. Finally the thread configuration mapping is updated in that, on the one hand, it gets extended regarding thread identifier n_2 and, on the other hand, the entry of thread identifier n_1 is updated.

$$[\textsc{Ass}]\ \frac{\begin{array}{c}\mathbf{tc}(n).cs = (\mu, x = e; mc) \circ \mathsf{CS}^b \\ (\mathsf{v}', \mu') = vupd(\mathsf{v}, \mu, x \mapsto [\![e]\!]_h^{\mathsf{v},\mu}) \quad \mathsf{CS} = (\mu', mc) \circ \mathsf{CS}^b\end{array}}{(h, \mathsf{v}, \mathbf{tc}) \rightsquigarrow (h, \mathsf{v}', \mathbf{tc}[n \mapsto \mathsf{CS}])}$$

$$[\textsc{FUpd}]\ \frac{\begin{array}{c}\mathbf{tc}(n).cs = (\mu, f = e; mc) \circ \mathsf{CS}^b \quad \mathsf{CS} = (\mu, mc) \circ \mathsf{CS}^b \\ o = [\![\mathtt{this}]\!]_h^{\mathsf{v},\mu} \quad (C, \mathsf{F}) = h(o) \quad h' = h[o \mapsto (C, \mathsf{F}[f \mapsto [\![e]\!]_h^{\mathsf{v},\mu}])]\end{array}}{(h, \mathsf{v}, \mathbf{tc}) \rightsquigarrow (h', \mathsf{v}, \mathbf{tc}[n \mapsto \mathsf{CS}])}$$

$$[\textsc{Call}]\ \frac{\begin{array}{c}\mathbf{tc}(n) = (C_T,\ (\mu, x = e.m(\overline{e}); mc) \circ \mathsf{CS}^b) \\ \mathsf{CS} = (\mathsf{v}_l, mbody(C, m)) \circ (\mu, \mathtt{rcv}\ x;\ mc) \circ \mathsf{CS}^b \\ o = [\![e]\!]_h^{\mathsf{v},\mu} \quad C = h(o).class \quad \overline{T}\,\overline{x} = mparams(C, m) \quad \overline{T_l}\,\overline{x_l} = mvars(C, m) \\ \mathsf{v}_l = \{\mathtt{this} \mapsto o, \mathtt{tid} \mapsto n, \mathtt{tclass} \mapsto C_T, \overline{x} \mapsto [\![\overline{e}]\!]_h^{\mathsf{v},\mu}, \overline{x_l} \mapsto ival(\overline{T_l})\}\end{array}}{(h, \mathsf{v}, \mathbf{tc}) \rightsquigarrow (h, \mathsf{v}, \mathbf{tc}[n \mapsto \mathsf{CS}])}$$

$$[\textsc{New}]\ \frac{\begin{array}{c}\mathbf{tc}(n).cs = (\mu, x = \mathtt{new}\ C(\overline{e}); mc) \circ \mathsf{CS}^b \\ \mathsf{CS} = (\mathsf{v}_l, cbody(C);\ \mathtt{return\ this}) \circ (\mu, \mathtt{rcv}\ x;\ mc) \circ \mathsf{CS}^b \\ o \in N \setminus dom(h) \quad h' = h[o \mapsto Obj_{\perp}^{C}] \quad \overline{T}\,\overline{x} = cparams(C) \quad \overline{T_l}\,\overline{x_l} = cvars(C) \\ \mathsf{v}_l = \{\mathtt{this} \mapsto o, \overline{x} \mapsto [\![\overline{e}]\!]_h^{\mathsf{v},\mu}, \overline{x_l} \mapsto ival(\overline{T_l})\}\end{array}}{(h, \mathsf{v}, \mathbf{tc}) \rightsquigarrow (h', \mathsf{v}, \mathbf{tc}[n \mapsto \mathsf{CS}])}$$

$$[\textsc{BlkBeg}]\ \frac{\begin{array}{c}\mathbf{tc}(n).cs = (\mu, \{\overline{T}\,\overline{x}; stmt\};\ mc) \circ \mathsf{CS}^b \quad \mathsf{CS} = (\mathsf{v}_l{\cdot}\mu, stmt;\ \mathtt{BE}\ mc) \circ \mathsf{CS}^b \\ \mathsf{v}_l = \{\overline{x} \mapsto ival(\overline{T})\}\end{array}}{(h, \mathsf{v}, \mathbf{tc}) \rightsquigarrow (h, \mathsf{v}, \mathbf{tc}[n \mapsto \mathsf{CS}])}$$

$$[\textsc{BlkEnd}]\ \frac{\mathbf{tc}(n).cs = (\mathsf{v}_l{\cdot}\mu, \mathtt{BE}\ mc) \circ \mathsf{CS}^b \quad \mathsf{CS} = (\mu, mc) \circ \mathsf{CS}^b}{(h, \mathsf{v}, \mathbf{tc}) \rightsquigarrow (h, \mathsf{v}, \mathbf{tc}[n \mapsto \mathsf{CS}])}$$

$$[\textsc{Whl}_1]\ \frac{\begin{array}{c}\mathbf{tc}(n).cs = (\mu, \mathtt{while}\ (e)\ \{stmt\}; mc) \circ \mathsf{CS}^b \\ \mathsf{CS} = (\mu, stmt;\ \mathtt{while}\ (e)\ \{stmt\}; mc) \circ \mathsf{CS}^b \quad [\![e]\!]_h^{\mathsf{v},\mu}\end{array}}{(h, \mathsf{v}, \mathbf{tc}) \rightsquigarrow (h, \mathsf{v}, \mathbf{tc}[n \mapsto \mathsf{CS}])}$$

$$[\textsc{Whl}_2]\ \frac{\mathbf{tc}(n).cs = (\mu, \mathtt{while}\ (e)\ \{stmt\}; mc) \circ \mathsf{CS}^b \quad \mathsf{CS} = (\mu, mc) \circ \mathsf{CS}^b \quad \neg[\![e]\!]_h^{\mathsf{v},\mu}}{(h, \mathsf{v}, \mathbf{tc}) \rightsquigarrow (h, \mathsf{v}, \mathbf{tc}[n \mapsto \mathsf{CS}])}$$

Table 6.4: *CoJapl* language: operational semantics (internal, part 1)

Also the interface communication rules of the operational semantics basically result from the rules of Table 2.12 by exchanging the configuration's call stack with a thread configuration mapping. Additionally, we extend the communication labels a concerning incoming and outgoing calls and returns with the thread n

$$[\textsc{Cond}_1]\ \frac{\begin{array}{c}\mathbf{tc}(n).cs = (\mu, \texttt{if}\ (e)\ \{stmt_1\}\ \texttt{else}\ \{stmt_2\};\ mc) \circ \mathsf{CS}^b \\ \mathsf{CS} = (\mu, stmt_1;\ mc) \circ \mathsf{CS}^b \quad [\![e]\!]_h^{\mathsf{v},\mu}\end{array}}{(h, \mathsf{v}, \mathbf{tc}) \rightsquigarrow (h, \mathsf{v}, \mathbf{tc}[n \mapsto \mathsf{CS}])}$$

$$[\textsc{Cond}_2]\ \frac{\begin{array}{c}\mathbf{tc}(n).cs = (\mu, \texttt{if}\ (e)\ \{stmt_1\}\ \texttt{else}\ \{stmt_2\};\ mc) \circ \mathsf{CS}^b \\ \mathsf{CS} = (\mu, stmt_2;\ mc) \circ \mathsf{CS}^b \quad \neg[\![e]\!]_h^{\mathsf{v},\mu}\end{array}}{(h, \mathsf{v}, \mathbf{tc}) \rightsquigarrow (h, \mathsf{v}, \mathbf{tc}[n \mapsto \mathsf{CS}])}$$

$$[\textsc{Ret}]\ \frac{\begin{array}{c}\mathbf{tc}(n).cs = (\mu_1, \texttt{return}\ e) \circ (\mu_2, \texttt{rcv}\ x;\ mc) \circ \mathsf{CS}^b \\ \mathsf{CS} = (\mu_2', mc) \circ \mathsf{CS}^b \quad (\mathsf{v}', \mu_2') = vupd(\mathsf{v}, \mu_2, x \mapsto [\![e]\!]_h^{\mathsf{v},\mu_2})\end{array}}{(h, \mathsf{v}, \mathbf{tc}) \rightsquigarrow (h, \mathsf{v}', \mathbf{tc}[n \mapsto \mathsf{CS}])}$$

$$[\textsc{Spawn}]\ \frac{\begin{array}{c}\mathbf{tc}(n_1).cs = (\mu, x = \texttt{spawn}\ C(\overline{e});\ mc) \circ \mathsf{CS}^b \quad \mathsf{CS}_1 = (\mu', mc) \circ \mathsf{CS}^b \\ n_2 \in N \setminus dom(\mathbf{tc}) \quad (\mathsf{v}', \mu') = vupd(\mathsf{v}, \mu, x \mapsto n_2) \quad \mathsf{CS}_2 = (\mathsf{v}_l, tbody(C)) \\ \overline{T}\ \overline{x} = tparams(C) \quad \mathsf{v}_l = \{\texttt{tid} \mapsto n_2, \texttt{tclass} \mapsto C, \overline{x} \mapsto [\![\overline{e}]\!]_h^{\mathsf{v},\mu}\}\end{array}}{(h, \mathsf{v}, \mathbf{tc}) \rightsquigarrow (h, \mathsf{v}', \mathbf{tc}[n_1 \mapsto \mathsf{CS}_1][n_2 \mapsto \mathsf{CS}_2])}$$

Table 6.5: *CoJapl* language: operational semantics (internal, part 2)

that carries out the communication step:

$$a ::= \gamma? \mid \gamma!$$
$$\gamma ::= n\langle call\ o.m(\overline{v})\rangle \mid n\langle new\ C(\overline{v})\rangle \mid n\langle return\ (v)\rangle \mid \nu(\Delta, \Theta).\gamma,$$

where $o \in N, v \in Vals$ and where Δ and Θ are type mappings.

To understand the reason for the extension of the labels, recall that a communication label shall consist of exactly the information that is passed to the receiver by the corresponding communication step. Now if, for instance, an incoming method call occurs, then the program does not only recognize the method m, the callee o, and the actual parameters $\overline{v}$ of the call but it can also find out the corresponding thread identifier by means of the expression `tid`. The same applies to constructor calls and to returns.

Note, in particular, that the thread of an incoming call may show up for the first time. Hence, the thread identifier may be included in the type mapping of the ν-binder. Since the program may inquire the thread class via `tclass`, such a new thread is typed with its class name. In contrast to the ν-binder of the sequential setting, the ν-binder of the multi-threaded setting consists of two mappings, representing the assumed and the committed types. For, in *Japl* an interface communication may only update either the commitment context or the assumption context. In *CoJapl* this is not always the case, anymore. The reason will become clear soon.

Apart from the program configuration modifications and from the above mentioned label extension, we have to deal with a new kind of interface communication, namely *cross-border thread spawning*. More specifically, the program may spawn a thread of an externally defined thread class provoking an *outgoing thread spawn label*. Likewise, the environment may spawn a thread concerning a thread class of the program resulting in an *incoming thread spawn label*. Again, the justification for dedicated labels regarding thread spawning is that the spawn is obviously an observable interaction: the new thread itself is aware of the fact that it just has been spawned. In order to find out the constituents of a spawn label, let us assume that the program spawns a new thread of an externally defined thread class. Such a spawn is certainly implemented in terms of a spawn statement

$$x = \texttt{spawn}\ C(\overline{e}),$$

where we consider C to be an externally defined thread class, i.e., a thread class of the program's environment. Similar to the cross-border constructor call, the name of the class and the actual parameters are part of the communication label. In contrast to a constructor call, where the calling thread is blocked until the environment yields the new object name, a thread spawn immediately returns and, thus, immediately yields the new thread identifier. As a consequence, the outgoing spawn label is equipped with the new thread identifier, such that the communication step provides both the communication partners with the new identifier. Symmetrically, an incoming spawn label includes the new thread identifier, as well. Therefore, we extend the above communication label definition as follows:

$$\gamma ::= \langle \textit{spawn}\, n\ \textit{of}\ C(\overline{v})\rangle.$$

Note that the spawn label γ provides the identifier n of the newly created thread only but not the thread identifier of the thread that has executed the `spawn` statement, as it is unknown to the new thread and, thus, unknown to the receiver of the communication step.

Now let us get back to the ν-binder. In the sequential setting a new name communicated in terms of an *incoming* communication represents always an object of an *environment* class, that is, the ν-binder of *incoming* communication always consists of an *assumption* type mapping Δ', only – objects of program classes are always created by the program itself. We have just seen, however, that regarding the multi-threaded setting an *incoming spawn* provides the identifier of the new thread already, even though the thread class is part of the *program*. Consequently, the thread identifier of the incoming spawn is typed with the program class such that the ν-binder includes a commitment type mapping Θ'. The parameters e of the spawn, yet again, may entail the propagation of new environment objects as well, thus, the spawn label is equipped with, both, a commitment *and* an assumption type context.

After this general introduction, the interface communcation rules of the operational semantics are given in Table 6.6. Similar to the internal computation

$$[\textsc{SpawnO}]\ \frac{\begin{array}{c} a = \nu(\Theta', \Delta').\langle spawn\ n\ of\ C(\overline{v})\rangle! \quad C \in dom(\Delta) \\ \mathbf{tc}(n') = (\mu, x{=}\mathtt{spawn}\ C(\overline{e}); mc) \circ \mathsf{CS} \\ \mathbf{tc}' = \mathbf{tc}[n \mapsto (\mu', mc) \circ \mathsf{CS}] \end{array}}{\Delta, \Delta' \vdash (h, \mathsf{v}, \mathbf{tc}) : \Theta \xrightarrow{a} \Delta \vdash (h, \mathsf{v}', \mathbf{tc}') : \Theta, \Theta'}$$

where $\overline{v} = [\![\overline{e}]\!]_h^{\mathsf{v},\mu}$, $n \in N \setminus dom(\mathbf{tc})$, $(\mathsf{v}', \mu') = vupd(\mathsf{v}, \mu, x \mapsto n)$, $\Delta' = (n{:}C)$, and $\Theta' = \mathsf{new}(h, \overline{v}, \Theta)$

$$[\textsc{SpawnI}]\ \frac{\begin{array}{c} a = \nu(\Delta', \Theta').\langle spawn\ n\ of\ C(\overline{v})\rangle? \quad \Delta \vdash a : \Theta \\ \mathbf{tc}' = \mathbf{tc}[n \mapsto (C, (\mathsf{v}_l, tbody(C))] \end{array}}{\Delta \vdash (h, \mathsf{v}, \mathbf{tc}) : \Theta \xrightarrow{a} \Delta, \Delta' \vdash (h, \mathsf{v}, \mathbf{tc}') : \Theta, \Theta'}$$

where $\overline{T}\,\overline{x} = tparams(C)$ and $\mathsf{v}_l = \{\mathtt{tid} \mapsto n, \mathtt{tclass} \mapsto C, \overline{x} \mapsto \overline{v}\}$

$$[\textsc{CalliI}]\ \frac{\begin{array}{c} a = \nu(\Delta').n\langle call\ o.m(\overline{v})\rangle? \quad \Delta \vdash a : \Theta \\ \mathbf{tc}(n) = (C_T, \mathsf{CS}^{eb}) \\ \mathbf{tc}' = \mathbf{tc}[n \mapsto (\mathsf{v}_l, mbody(C, m)) \circ \mathsf{CS}^{eb}] \end{array}}{\Delta \vdash (h, \mathsf{v}, \mathbf{tc}) : \Theta \xrightarrow{a} \Delta, \Delta' \vdash (h, \mathsf{v}, \mathbf{tc}') : \Theta}$$

where $\Theta \vdash o : C$, $\overline{T}\,\overline{x} = mparams(C, m)$, $\overline{T'}\,\overline{x'} = mvars(C, m)$, and $\mathsf{v}_l = \{\mathtt{tid} \mapsto n, \mathtt{tclass} \mapsto C_T, \mathtt{this} \mapsto o, \overline{x} \mapsto \overline{v}, \overline{x'} \mapsto ival(\overline{T'})\}$

$$[\textsc{NewI}]\ \frac{\begin{array}{c} a = \nu(\Delta').n\langle new\ C(\overline{v})\rangle? \quad \Delta \vdash a : \Theta \\ \mathbf{tc}(n) = (C_T, \mathsf{CS}^{eb}) \\ \mathbf{tc}' = \mathbf{tc}[n \mapsto (\mathsf{v}_l, cbody(C)) \circ \mathsf{CS}^{eb}] \end{array}}{\Delta \vdash (h, \mathsf{v}, \mathbf{tc}) : \Theta \xrightarrow{a} \Delta, \Delta' \vdash (h', \mathsf{v}, \mathbf{tc}') : \Theta}$$

where $o \in N \setminus dom(h)$, $h' = h[o \mapsto Obj_{\perp}^{C}]$, $\overline{T}\,\overline{x} = cparams(C)$, $\overline{T'}\,\overline{x'} = cvars(C)$, and $\mathsf{v}_l = \{\mathtt{tid} \mapsto n, \mathtt{tclass} \mapsto C_T, \mathtt{this} \mapsto o, \overline{x} \mapsto \overline{v}, \overline{x'} \mapsto ival(\overline{T'})\}$

$$[\textsc{CalliI}_{nt}]\ \frac{\begin{array}{c} a = \nu(\Delta').n\langle call\ o.m(\overline{v})\rangle? \quad \Delta \vdash a : \Theta \\ \Delta' \vdash n : C_T \\ \mathbf{tc}' = \mathbf{tc}[\ n \mapsto (\ C_T, \mathsf{v}_l, mbody(C, m))\ )\ ] \end{array}}{\Delta \vdash (h, \mathsf{v}, \mathbf{tc}) : \Theta \xrightarrow{a} \Delta, \Delta' \vdash (h, \mathsf{v}, \mathbf{tc}') : \Theta}$$

where $\Theta \vdash o : C$, $\overline{T}\,\overline{x} = mparams(C, m)$, $\overline{T'}\,\overline{x'} = mvars(C, m)$, and $\mathsf{v}_l = \{\mathtt{tid} \mapsto n, \mathtt{tclass} \mapsto C_T, \mathtt{this} \mapsto o, \overline{x} \mapsto \overline{v}, \overline{x'} \mapsto ival(\overline{T'})\}$

$$[\textsc{NewI}_{nt}]\ \frac{\begin{array}{c} a = \nu(\Delta').n\langle new\ C(\overline{v})\rangle? \quad \Delta \vdash a : \Theta \\ \Delta' \vdash n : C_T \\ \mathbf{tc}' = \mathbf{tc}[\ n \mapsto (\ C_T, (\mathsf{v}_l, cbody(C))\ )\ ] \end{array}}{\Delta \vdash (h, \mathsf{v}, \mathbf{tc}) : \Theta \xrightarrow{a} \Delta, \Delta' \vdash (h', \mathsf{v}, \mathbf{tc}') : \Theta}$$

where $o \in N \setminus dom(h)$, $h' = h[o \mapsto Obj_{\perp}^{C}]$, $\overline{T}\,\overline{x} = cparams(C)$, $\overline{T'}\,\overline{x'} = cvars(C)$, and $\mathsf{v}_l = \{\mathtt{tid} \mapsto n, \mathtt{tclass} \mapsto C_T, \mathtt{this} \mapsto o, \overline{x} \mapsto \overline{v}, \overline{x'} \mapsto ival(\overline{T'})\}$

Table 6.6: *CoJapl* language: operational semantics (external)

rules, most of the external rules are similar to their sequential counterparts of Table 2.12. As mentioned above, however, we additionally introduce two rules concerning incoming spawns and, respectively, outgoing spawns. Rule SPAWNO deals with a `spawn` statement that results in an outgoing spawn label, i.e., the corresponding thread class C is in the domain of the assumption context Δ. We said already that the ν-binder of a spawn label provides an assumption and an commitment update context. The commitment context Θ' is determined by means of the auxiliary function new introduced in Section 2.4.3. The assumption context consists exactly of the thread identifier n which is used for the newly spawned thread.

Dually, Rule SPAWNI deals with an incoming spawn label. The domain of the thread configuration mapping is extended by the thread class name C and a new call stack consisting of the thread class's thread body.

The environment can create threads by means of externally defined thread classes, hence, the corresponding thread creation process is not observable by the program. As a consequence, an incoming call via a new thread may occur, i.e., a thread which is unknown to the program so far. On account of this, the operational semantics of *CoJapl* provides two rules for incoming method calls and, respectively, for incoming constructor calls. Rules CALLI and NEWI deal with incoming calls by means of a thread n which is known to the program already. In particular, the rules' original thread configuration mapping **tc** maps n to a thread class C_T and a call stack. Thus, the incoming call causes an extension of the existing call stack by a new activation record.

On the other hand, Rule CALLI_{nt} and Rule NEWI_{nt} are used for incoming calls via an unknown thread n. The novel character of n is indicated by the fact that n is bound in the communication label a such that is included in the label's type context Δ'. Consequently, the thread configuration mapping is extended by the new thread n, where n is mapped to its thread class and a new call stack consisting of the method or, respectively, the constructor body of the call.

Note, in contrast to the previous incoming communication rules, the three new rules SPAWNI, CALLI_{nt}, and NewI_{nt} dealing with new threads do not require an externally blocked call stack in the original configuration.

We conclude this section with a definition of the program execution, the initial configurations, and the trace semantics of a *CoJapl* program. It is no big surprise that these definitions resemble the corresponding definitions of the sequential language.

Definition 6.3.1 (Execution, initial configurations, and trace semantics)**:** Let

$$p \equiv \overline{impdecl};\ \overline{T}\,\overline{x};\ \overline{cldef}\ \overline{tdef}\ \{stmt;\ \texttt{return}\}$$

be a syntactically correct and well-typed *CoJapl* program. A *program execution* of p is a finite sequence of reduction steps, according to the rules of Table 6.4, Table 6.5, and

$$[\textsc{Intern}]\ \frac{c \rightsquigarrow^* c'}{\Delta \vdash c : \Theta \overset{\epsilon}{\Longrightarrow} \Delta' \vdash c' : \Theta'}$$

$$[\textsc{Single}]\ \frac{\Delta \vdash c : \Theta \overset{a}{\longrightarrow} \Delta' \vdash c' : \Theta'}{\Delta \vdash c : \Theta \overset{a}{\Longrightarrow} \Delta' \vdash c' : \Theta'}$$

$$[\textsc{Seqnc}]\ \frac{\Delta \vdash c : \Theta \overset{s}{\Longrightarrow} \Delta' \vdash c' : \Theta' \qquad \Delta' \vdash c' : \Theta' \overset{t}{\Longrightarrow} \Delta'' \vdash c'' : \Theta''}{\Delta \vdash c : \Theta \overset{st}{\Longrightarrow} \Delta'' \vdash c'' : \Theta''}$$

Table 6.7: *CoJapl* language: traces

Table 6.6, starting from an *initial configuration* of the program

$$\begin{aligned} c_{init}(p) \overset{\text{def}}{=} (\ & h_\perp, \{\overline{x} \mapsto ival(\overline{T})\}, \\ & \{\ n_{main} \mapsto (\ \{\texttt{tid} \mapsto n_{main},\ \texttt{tclass} \mapsto \mathit{Main}\},\ \mathit{stmt};\ \texttt{return}\ )\ \}\ ), \end{aligned}$$

or, respectively,

$$\overline{c_{init}}(p) \overset{\text{def}}{=} (h_\perp, \{\overline{x} \mapsto ival(\overline{T})\},\ \epsilon).$$

Correspondingly, by means of the rules of Table 6.7, we define three semantic functions

$$[\![\cdot]\!]^a_{trace}, [\![\cdot]\!]^p_{trace}, [\![\cdot]\!] : \Delta \vdash p : \Theta \rightharpoonup \mathscr{P}(a^*),$$

such that for $\Delta \vdash p : \Theta$ it is

$$\begin{aligned} [\![\Delta \vdash p : \Theta]\!]^a_{trace} &\overset{\text{def}}{=} \{s \in a^* | \Delta \vdash c_{init}(p) : \Theta \overset{s}{\Longrightarrow} \Delta' \vdash c' : \Theta'\}, \\ [\![\Delta \vdash p : \Theta]\!]^p_{trace} &\overset{\text{def}}{=} \{s \in a^* | \Delta \vdash \overline{c_{init}}(p) : \Theta \overset{s}{\Longrightarrow} \Delta' \vdash c' : \Theta'\}, \text{ and} \\ [\![\Delta \vdash p : \Theta]\!] &\overset{\text{def}}{=} [\![\Delta \vdash p : \Theta]\!]^a_{trace} \cup [\![\Delta \vdash p : \Theta]\!]^p_{trace}. \end{aligned}$$

Chapter 7

Test specification language and code generation

As in the sequential setting, the underlying idea of our testing approach also in the multi-threaded setting is to provide a test specification language which allows to specify the *interface interactions* that may occur between the component under test and its environment. Regarding the sequential setting, the specification language's look-and-feel resembles that of the programming language *Japl* but also the specification language's semantics was geared towards *Japl*'s trace semantics. Describing a desired interaction trace, i.e., a *sequence* of communication labels, a *Japl* test specification, in particular, specifies the exact order of the entailed interface interactions. The consequence is that, in *Japl* we only need a single, sequentially composed, (main) specification statement which likewise stipulates an exact order due to its sequential construction.

In the multi-threaded setting, a trace of the semantics also represents a sequence of interactions. Due to the non-deterministic scheduling policy of the language, however, we cannot assure a certain sequence in general: if a program realizes a certain trace, then it also realizes different possible interleavings of the original trace. On account of this, we want to allow for specifying tests that are relaxed regarding the order of interactions carried out by different threads. Interactions that belong to the same thread, however, must again comply with a certain order. Therefore the idea is that the concurrent specification language shall allow to provide a specification statement *for each* thread that becomes active in the specification. To achieve this, we first have to identify the different situations in which a thread (identifier) may show up in a *CoJapl* program for the first time. There exist four different ways which are:

- internal thread creations due to instantiation of a thread class of the program

itself; for instance:

$$x = \texttt{spawn}\ \mathit{MyThread}(e, \ldots, e),$$

- outgoing spawn labels, that is, the program instantiates an external thread class; for instance:

$$a = \nu(\Delta', \Theta').\langle \mathit{spawn}\ n\ \mathit{of}\ C(\overline{v})\rangle!,$$

- incoming spawn labels, that is, the program's environment instantiates a thread class of the program, resulting in a communication label, like:

$$a = \nu(\Delta', \Theta').\langle \mathit{spawn}\ n\ \mathit{of}\ C(\overline{v})\rangle?,$$

- and incoming method or constructor call labels where the call is carried out by a new thread; for instance:

$$a = \nu(\Delta').n\langle \mathit{call}\ o.m(\overline{v})\rangle?, \quad \text{such that } n \in \Delta'.$$

For each of the above listed situations, a specification must provide a corresponding specification statement that determines the desired sequence of interface interaction carried out by the new thread. As in the single threaded case, we want to accomplish this by extending the programming language *CoJapl* with dedicated specification constructs.

Let us start with a spawn statement that results in an outgoing spawn label. That is, the specification instantiates a thread class C of the component under test. Following the style of the outgoing call statement of the sequential specification language, the spawn statement of the multi-threaded specification language resembles the original spawn statement but is equipped with an exclamation mark to indicate the cross-border communication.

$$x = \texttt{spawn!}C(e, \ldots, e).$$

A crucial difference between a method call specification statement (as well as a constructor call specification statement) on one hand and the spawn specification statement on the other hand is that the spawn statement is not split into two parts at the equal sign. For, the thread that carries out the spawn statement does not get blocked but always immediately returns such that the thread cannot realize any other actions in between the spawn and the corresponding return of the thread identifier. Since the spawn statement causes the creation of a new thread, the specification shall entail a description of the desired interface interactions realized by the new thread. To this end, we introduce the following *test thread construct* for specifying the interface behavior of a thread class C pertaining to the component under test:

$$\texttt{test thread}\ C(\overline{T}\,\overline{x})\{\ \mathit{stmt}\ \}.$$

According to our example, the above mentioned outgoing spawn statement results in a new thread of thread class C of the component under test and this new thread shall expose a behavior that conforms to the specification statement *stmt*. Note that *stmt* is parameterized regarding the spawn's parameters. Also note that the statement will be typed in a passive control context, as C is defined within the external component, hence, the thread becomes active in the external component, as well.

It has been mentioned, however, that a thread of an external thread class can also be created by the external component itself such that the specification does not know anything about the thread until it passes the interface for the very first time due to an incoming method or constructor call. In these cases the specification has to provide a desired behavior for the new thread, as well. Though, the corresponding specification statement cannot be parameterized regarding the spawn parameters, as it was not the specification that causes the thread spawning but the external component, hence, the corresponding actual parameters are not known to the specification. Therefore, we introduce a second test thread construct for specifying the behavior of thread classes of the component under test. In particular, it is almost identical to the aforementioned parameterized thread specification construct except that it does not provide any parameters. Therefore, the specification construct

$$\texttt{test thread}\ C\{\ \mathit{stmt}\ \}$$

means that, if a thread of class C enters the specification via a method or constructor call for the first time, then the interface behavior realized by this thread has to comply with the specification statement *stmt*.

Similar to the *CoJapl* programming language, a specification may not only spawn threads of externally defined thread classes but it also may create threads by means of internal thread creations, that is, the specification also supports the following spawn statement:

$$x = \texttt{spawn}(e, \ldots, e).$$

Note, however, in contrast to a *CoJapl* program, a specification may only use this statement in order to realize internal thread creations, since external thread creations are implemented by the above mentioned outgoing spawn statement.

A thread that comes to existence due to an internal thread creation also has to stick to a specified interface interaction sequence. Therefore, the specification language also provides a *mock thread construct* that stipulates a certain interface interaction due to a thread of a specification thread class C:

$$\texttt{mock thread}\ C(\overline{T}\ \overline{x})\{\ \mathit{stmt}\ \}.$$

Since a thread of a specification thread class C always starts in the specification, it consequently may pass the interface for the first time due to an outgoing communication, only. Hence, the specification statement *stmt* is typed in an active control context.

A thread of a specification thread class can come into existence either due to an internal spawn statement or due to an incoming spawn label. In both cases, the corresponding thread creation parameters are observable to the specification. Therefore, regarding specification thread classes, we do not need an additional mock thread construct that lacks the parameters.

Note, furthermore, that we need not to provide a specification statement for the expectation of *incoming* spawns. To understand the reason, consider the case that we do provide such a specification statement. More specifically, assume a specification of a thread n which entails the following fictitious incoming spawn statement:

$$x = \texttt{spawn}?C(\overline{T}\,\overline{x}).\texttt{where}(e).$$

The statement specifies that we expect the component under test to provoke an incoming spawn label via thread n resulting in a new thread. Let us assume, the name of the new thread is n'. Within the thread n itself, however, we cannot check if a spawn was executed. Also the new thread n' cannot verify that it was created by the specified spawn, as the originator of a spawn is unknown to the new thread, in general. Finally, due to the scheduling policy, the execution of a spawn statement and the resulting execution of the corresponding thread body are *decoupled* such that the start of the new thread does not allow to infer the point of time, when the spawn statement has been executed. For instance, even if the component under test executes, as specified by the above spawn expectation statement, the right spawn statement at the desired point of time, then the new thread n' may be scheduled much later. The conclusion is, neither can the specification observe the originator nor the point of time regarding an incoming spawn rendering it useless to introduce a corresponding specification statement.

Summarizing, the above mentioned mock thread construct is used for, both, internally spawned and externally spawned threads of specification thread classes. Now that we have discussed the new specification construct, the following section provides the syntax of the concurrent specification language, at large.

7.1 Syntax

The syntax of the test specification language for testing *CoJapl* components is given in terms of a grammar definition in Table 7.1. The language is basically an extension of the specification language, given in Table 3.1, by the interactions specification of thread classes. To this end, the language provides the new constructs that we have introduced in the previous section. In particular, the test thread constructs extend the declaration of the test unit classes while the mock thread construct completes the mock class declarations. We assume that the thread class names of all kinds of thread specification constructs are different. Note that, due to simplicity, this also means that the behavior of a thread class of the component under test may only be specified *either* by means of a parameterized test thread specification construct *or* by the non-parameterized test thread construct. Consequently, we assume that all instances of an external thread class may show up

s ::=	$\overline{cutdecl};\ \overline{mokdecl};\ \overline{T}\,\overline{x};\ \{\ stmt\ \}$	specification
$cutdecl$ ::=	$\texttt{test class}\ C$	test unit classes
	$\mid \texttt{test thread}\ C(\overline{T}\,\overline{x})\{\ stmt\ \}$	
	$\mid \texttt{test thread}\ C\{\ stmt\ \}$	
$mokdecl$ ::=	$\texttt{mock class}\ C\{C(T,\ldots,T);\ \overline{T\ m(T,\ldots,T)}\}$	mock classes
	$\mid \texttt{mock thread}\ C(\overline{T}\,\overline{x})\{\ stmt\ \}$	
$stmt$ ::=	$x = e \mid x = \texttt{new}\ C \mid \varepsilon \mid stmt; stmt \mid \{\overline{T}\,\overline{x};\ stmt\}$	statements
	$\mid \texttt{while}\ (e)\ \{stmt\} \mid \texttt{if}\ (e)\ \{stmt\}\ \texttt{else}\ \{stmt\}$	
	$\mid x = \texttt{spawn}\ C(e,\ldots,e)$	
	$\mid stmt_{in} \mid stmt_{out} \mid \texttt{case}\ \{\ \overline{stmt_{in}; stmt}\ \}$	
$stmt_{in}$ ::=	$(C\,x)?m(\overline{T}\,\overline{x}).\texttt{where}(e)\ \{\overline{T}\,\overline{x};\ stmt;\ !\texttt{return}\ e\}$	incoming stmt
	$\mid \texttt{new}(C\,x)?C(\overline{T}\,\overline{x}).\texttt{where}(e)\{\overline{T}\,\overline{x};\ stmt;\ !\texttt{return}\}$	
$stmt_{out}$ ::=	$e!m(e,\ldots,e)\ \{\overline{T}\,\overline{x};\ stmt;\ ?\texttt{return}(x).\texttt{where}(e)\ \}$	outgoing stmt
	$\mid \texttt{new}!C(e,\ldots,e)\{\overline{T}\,\overline{x};\ stmt;\ ?\texttt{return}(x).\texttt{where}(e)\}$	
	$\mid x = \texttt{spawn}!C(e,\ldots,e)$	
e ::=	$x \mid \texttt{null} \mid \texttt{op}(e,\ldots,e) \mid \texttt{tid} \mid \texttt{tclass}$	expressions

Table 7.1: Specification language for *CoJapl*: syntax

for the first time either due to an outgoing spawn or due to an incoming call.

Finally, the set of statements is extended by the internal and the outgoing spawn specification. Regarding the expressions, like in *CoJapl* we introduce the new expressions `tid` and `tclass` which allow a thread to determine its identifier and its class, respectively.

We conclude this section with two small examples which illustrate the usage of the specification language. The examples are given in Table 7.2. The first example on the left hand side of the table demonstrates the behavior specification of externally defined thread class, i.e., thread classes provided by the component under test. We assume that the component under test implements a thread that communicates with a (simplified) network service via its socket API (cf. [72], for instance). The used socket API, however, is wrapped in a *ServerSocket* class which is mocked by the specification. The thread under test is spawned by the main statement of the specification in Line 33. The thread specification is given in Lines 7 to 31. Specifically, the component (more precisely: its thread) is expected to create a *ServerSocket* object. Afterwards the component shall request the socket to listen to the network by means of an invocation of method *listen*. When the socket gets a connection request from the network it returns from the *listen* call. In this example, the method immediately return simulating a connection request. The component, in turn, accepts the connection by calling *accept* which is then followed by an undetermined number of *send* requests and by a final call of the *close* method.

The second example in Table 7.2 illustrates the specification of a mock thread class *StackTest* (Lines 2 to 26). It assumes an externally defined *Stack* class. More

specifically, the thread class tests if the *Stack* implementation is *thread-save*. That is, instances of the *Stack* class used via different threads must not interfere with each other resulting in an invalid stack structure. The thread class is parametrized in terms of three integer values which are pushed to and afterwards popped from a *Stack* object. Finally, the main specification statement spawns three threads of *StackTest* which, therefore, are executed concurrently.

Server socket example:

```
mock class ServerSocket{
  ServerSocket ServerSocket();
  bool listen();
  ...
}

test thread ServSockThread() {
  ServerSocket s;

  new?(ServerSocket x)ServerSocket() {
    s = x;
    !return(s)
  };
  s?listen() {
    !return(true)
  };
  s?accept() {
    !return(true);
  };
  bool rcv = true;
  while (rcv) {
    case
      s?send(Data d) {
        !return(true)
        }
      s?close() {
        rcv = false;
        !return(true)
        }
  }
}
{ thread x;
  x = spawn!servSockThread()
}
```

Stack example:

```
test class Stack;
mock thread StackTest(int x1, x2, x3) {
  Stack s;

  new!Stack() {
    ?return(s)
  };
  s!push(x1) {
    ?return(1)
  };
  s!push(x2) {
    ?return(2)
  };
  s!push(x3) {
    ?return(3)
  };
  s!pop(){
    ?return(x3)
  }
  s!pop(){
    ?return(x2)
  }
  s!pop(){
    ?return(x1)
  }
}
{ thread x;
  x = spawn StackTest(1, 2, 3);
  x = spawn StackTest(4, 5, 6);
  x = spawn StackTest(7, 8, 9);
}
```

Table 7.2: *CoJapl* example specifications

7.2 Static semantics

The type system for the concurrent test specification language is given in Table 7.3. It extends the type system for the sequential language, given in Table 3.2, by new rules regarding the newly introduced constructs. Moreover, Rule T-SPEC requires a simple adaption, as the mock thread declarations have to be type-checked, while the mock class declarations of the sequential settings were only used to extract the type information.

Rule T-TESTT$_{Spwn}$ deals with the test thread specification of threads spawned by means of an outgoing spawn label. Thus, the corresponding thread class C has to be included in the assumption context. Moreover, its type must comply with the thread specification. Finally, the body statement *stmt* of the thread specification has to be well-typed regarding a type context that is enriched by the thread creation parameters. Specifically, the statement has to be a passive statement, since the thread starts within the component under test. Similarly, the Rule T-TESTT$_{Call}$ deals with the specification construct of an external thread that shows up in the specification due to an incoming call. Therefore, the specification construct is not parameterized by the thread creation parameters, so we can omit the type check of the previous rule. Yet, also here, the name C must be typed as an externally defined thread class. Likewise, the specification statement must be passive.

$$[\text{T-SPEC}]\ \frac{\Theta = \mathit{cltype}(\overline{\mathit{mokdecl}}) \quad \Gamma;\Delta;\Theta \vdash \overline{\mathit{cutdecl}} : \mathsf{ok} \quad \Gamma;\Delta;\Theta \vdash \overline{\mathit{mokdecl}} : \mathsf{ok} \quad \Gamma' = \Gamma, \overline{x}{:}\overline{T} \quad \Gamma';\Delta;\Theta \vdash \mathit{stmt} : \mathsf{ok}^{\gamma};}{\Gamma;\Delta \vdash \overline{\mathit{cutdecl}};\ \overline{\mathit{mokdecl}};\ \overline{T}\,\overline{x};\ \{\ \mathit{stmt}\ \} : \Theta^{\gamma}}$$

$$[\text{T-TESTT}_{Spwn}]\ \frac{\Delta \vdash C : \overline{T} \quad \Gamma' = \Gamma, \overline{x}{:}\overline{T} \quad \Gamma';\Delta;\Theta \vdash \mathit{stmt} : \mathsf{ok}^{psv}}{\Gamma;\Delta;\Theta \vdash \texttt{test thread}\ C(\overline{T}\,\overline{x})\{\ \mathit{stmt}\ \} : \mathsf{ok}}$$

$$[\text{T-TESTT}_{Call}]\ \frac{\Delta \vdash C : \overline{T} \quad \Gamma';\Delta;\Theta \vdash \mathit{stmt} : \mathsf{ok}^{psv}}{\Gamma;\Delta;\Theta \vdash \texttt{test thread}\ C\{\ \mathit{stmt}\ \} : \mathsf{ok}}$$

$$[\text{T-MOCKT}]\ \frac{\Gamma' = \Gamma, \overline{x}{:}\overline{T} \quad \Gamma';\Delta;\Theta \vdash \mathit{stmt} : \mathsf{ok}^{act}}{\Gamma;\Delta;\Theta \vdash \texttt{mock thread}\ C(\overline{T}\,\overline{x})\{\ \mathit{stmt}\ \} : \mathsf{ok}}$$

$$[\text{T-SPAWN}_i]\ \frac{\Gamma;\Delta,\Theta \vdash x{:}\texttt{thread} \quad \Theta \vdash C{:}\overline{T} \quad \Gamma;\Delta,\Theta \vdash \overline{e}{:}\overline{T}}{\Gamma;\Delta;\Theta \vdash x = \texttt{spawn}\ C(\overline{e}) : \mathsf{ok}^{act}}$$

$$[\text{T-SPAWN}_{out}]\ \frac{\Gamma;\Delta,\Theta \vdash x{:}\texttt{thread} \quad \Delta \vdash C{:}\overline{T} \quad \Gamma;\Delta,\Theta \vdash \overline{e}{:}\overline{T}}{\Gamma;\Delta;\Theta \vdash x = \texttt{spawn!}C(\overline{e}) : \mathsf{ok}^{act}}$$

Table 7.3: Specification language for *CoJapl*: type system (stmts)

The mock thread specification represents the dual of the test thread specification. Thus, its typing rule T-MOCKT is almost identical to Rule T-TESTT$_{spwn}$ but only its statement is type-checked in an active control context.

Finally, the two new spawn statements are type-checked by Rule SPAWN$_i$ and Rule SPAWN$_{out}$, respectively. In both cases, the variable x has to be a thread variable and the class name C must be appropriately typed as a thread class. Regarding the internal spawn statement, however, the thread class must be provided by the commitment context Θ while the outgoing spawn statement is only well-typed if the thread class can be found in the assumption context Δ. Note that both statements are only well-typed in an active control context.

7.3 Operational semantics

Again, the internal steps of the operational semantics are identical to the rules of the concurrent programming language *CoJapl*, hence, we do not repeat them again. Regarding the external steps, we adapt the rules of the sequential specification language by extending it with threads. This is done, as explained above, by exchanging the call stack of the configurations with a thread configuration mapping. Therefore, we also omit most of the rules inherited from the sequential setting. We add new rules for the new thread-related specification constructs. As with *CoJapl*, we have to differentiate incoming calls via new threads from rules regarding re-entrant threads. The new rules are shown in Table 7.4.

An incoming spawn causes the extension of the thread configuration mapping **tc**, where the new call stack is initialized with the specification statement of the corresponding mock thread specification. To this end, we redefine the code extracting function *cbody* such that it extracts the body statement from mock and test thread specifications. We omit the straightforward redefinition of *cbody*.

Similarly, an outgoing spawn causes the extension of **tc**, where the call stack is initialized with the specification statement of the corresponding test thread specification. Additionally, the call stack that implements the outgoing spawn statement is reduced and the global and local variables are updated with the new thread identifier.

Regarding incoming method and constructor calls, we have to provide two rules each, as explained above. Rule CALLI deals with incoming method calls realized by a thread n that is known to the specification, already. In particular, there exist a thread configuration for n in **tc** already, whose call stack specifies the expectation of this incoming call. Furthermore, the where-clause e' of the expectation evaluates to true, so the call stack is reduced and the thread configuration mapping is correspondingly updated.

Rule CALLI$_{nt}$, in contrast, deals with incoming calls that are realized by means of a new thread. Thus, the thread configuration mapping does not provide a corresponding mapping. However, the rules requires that a test thread specification regarding this thread is provided, such that the thread specification's first expectation statement matches with the incoming call. In this case, the thread configuration mapping is extended by a new thread configuration for the new

$$[\textsc{SpawnI}]\ \frac{\begin{array}{c} a = \nu(\Delta', \Theta').\langle \mathit{spawn}\ n\ \mathit{of}\ C(\overline{v})\rangle? \quad \Delta \vdash a : \Theta \\ \mathbf{tc}' = \mathbf{tc}[n \mapsto (C, (\mathsf{v}_l, \mathit{tbody}(C))] \end{array}}{\Delta \vdash (h, \mathsf{v}, \mathbf{tc}) : \Theta \xrightarrow{a} \Delta, \Delta' \vdash (h, \mathsf{v}, \mathbf{tc}') : \Theta, \Theta'}$$

where $\overline{T}\,\overline{x} = \mathit{tparams}(C)$ and $\mathsf{v}_l = \{\mathtt{tid} \mapsto n, \mathtt{tclass} \mapsto C, \overline{x} \mapsto \overline{v}\}$

$$[\textsc{SpawnO}]\ \frac{\begin{array}{c} a = \nu(\Theta', \Delta').\langle \mathit{spawn}\ n\ \mathit{of}\ C(\overline{v})\rangle! \\ \mathbf{tc}(n').cs = (\mu, x{=}\mathtt{spawn}!C(\overline{e}); mc) \circ \mathsf{CS} \\ \mathbf{tc}' = \mathbf{tc}[n' \mapsto (\mu', mc) \circ \mathsf{CS}][n \mapsto (C, (\mathsf{v}_l, \mathit{tbody}(C)))] \end{array}}{\Delta, \Delta' \vdash (h, \mathsf{v}, \mathbf{tc}) : \Theta \xrightarrow{a} \Delta \vdash (h, \mathsf{v}', \mathbf{tc}') : \Theta, \Theta'}$$

where $\overline{v} = [\![\overline{e}]\!]_h^{\mathsf{v},\mu}$, $n \in N \setminus \mathit{dom}(\mathbf{tc})$, $(\mathsf{v}', \mu') = \mathit{vupd}(\mathsf{v}, \mu, x \mapsto n)$, $\Delta' = (n{:}C)$, $\Theta' = \mathsf{new}(h, \overline{v}, \Theta)$, and $\mathsf{v}_l = \{\mathtt{tid} \mapsto n, \mathtt{tclass} \mapsto C, \overline{x} \mapsto \overline{v}\}$

$$[\textsc{CalleI}]\ \frac{\begin{array}{c} a = \nu(\Delta').n\langle \mathit{call}\ o.m(\overline{v})\rangle? \quad \Delta \vdash a : \Theta \\ s^{psv} = (C\ x)?m(\overline{T}\,\overline{x}).\mathtt{where}(e')\ \{\overline{T_l}\ \overline{x_l};\ s^{act};\ !\mathtt{return}\ e\} \\ [\![e']\!]_h^{\mathsf{v},\mathsf{v}_l \cdot \mu} \quad \mathbf{tc}(n).cs = (\mu, s^{psv};\ mc^{psv}) \circ \mathsf{CS} \\ \mathbf{tc}' = \mathbf{tc}[n \mapsto (\mathsf{v}_l{\cdot}\mu, s^{act};\ !\mathtt{return}(e);\ mc) \circ \mathsf{CS}^{eb}] \end{array}}{\Delta \vdash (h, \mathsf{v}, \mathbf{tc}) : \Theta \xrightarrow{a} \Delta, \Delta' \vdash (h, \mathsf{v}, \mathbf{tc}') : \Theta}$$

where $C = \Theta(o)$, $\Delta, \Theta \vdash n : C_T$, and $\mathsf{v}_l = \{\mathtt{tid} \mapsto n, \mathtt{tclass} \mapsto C_T, \mathtt{this} \mapsto o, \overline{x} \mapsto \overline{v}, \overline{x'} \mapsto \mathit{ival}(\overline{T'})\}$

$$[\textsc{CalleI}_{nt}]\ \frac{\begin{array}{c} a = \nu(\Delta').n\langle \mathit{call}\ o.m(\overline{v})\rangle? \quad \Delta' \vdash n : C_T \\ \Delta \vdash a : \Theta \quad \mathit{tbody}(C_T) = s^{psv};\ s_1^{psv} \\ s^{psv} = (C\ x)?m(\overline{T}\,\overline{x}).\mathtt{where}(e')\{\overline{T_l}\ \overline{x_l};\ s^{act};\ !\mathtt{return}\ e\} \\ [\![e']\!]_h^{\mathsf{v},\mathsf{v}_l} \quad \mathbf{tc}' = \mathbf{tc}[\ n \mapsto (\mathsf{v}_l, s^{act};\ !\mathtt{return}\ e;\ s_1^{psv})] \end{array}}{\Delta \vdash (h, \mathsf{v}, \mathbf{tc}) : \Theta \xrightarrow{a} \Delta, \Delta' \vdash (h, \mathsf{v}, \mathbf{tc}') : \Theta}$$

where $C = \Theta(o)$, and $\mathsf{v}_l = \{\mathtt{tid} \mapsto n, \mathtt{tclass} \mapsto C_T, \mathtt{this} \mapsto o, \overline{x} \mapsto \overline{v}, \overline{x'} \mapsto \mathit{ival}(\overline{T'})\}$

Table 7.4: Specification language for *CoJapl*: operational semantics (external)

thread where the call stack consists of the thread's specification statement. We skip the rules for incoming constructor call, as they are very similar to the rules for incoming method calls.

7.4 Test code generation

In this section we want to sketch a possible extension of the sequential code generation algorithm achieving a code generation algorithm for the multi-threaded setting.

Recall, that a central idea of the sequential test code generation was the *anticipation* of the next incoming communication. Specifically, in the sequential setting we annotated each incoming communication term of the specification with an expectation identifier. On the other hand, we added a global variable *next* which provided the label of the next upcoming incoming communication term. This way, we could distribute the (translated) code of the specification over several method definitions without loosing track of the stipulated sequential order of the specified interface interactions.

As for the multi-threaded setting, we will embark exactly on the same strategy, though the sequential order and the corresponding anticipation mechanism will be carried out for each thread, only. In particular, all specification statements of each mock thread and test thread specification are equipped with a unique expectation identifier annotation as well as with corresponding *next* update statements, so that, for instance, as a first approach, a mock thread specification

$$\texttt{mock thread}\ C(\overline{T}\,\overline{x})\{\ s^{act}\ \}$$

is preprocessed according to the preprocessing step as described in Section 4.1 resulting in a thread specification

$$\texttt{mock thread}\ C(\overline{T}\,\overline{x})\{\ prep_{out}(s^{psv})\ \},$$

where s^{psv}_{pp} results from applying $prep_{out}$ on s^{psv}. It is crucial, that each thread uses its own *next* variable since the specification stipulates the sequential order of interactions only per thread. In this context, a little complication arise from the fact that threads can be spawned dynamically. For, it is not sufficient to declare a static number of global *next* variables but instead we have to implement the *next* variables by means of a globally accessible dynamic list which maps thread identifiers to the corresponding next expectation identifier. We abstract the details regarding the list implementation away such that in the following we refer to the globally accessible list in terms of an array. That is, we assume that the preprocessed specification provides a dynamic list *next* where the expression $next[n]$ yields the next expectation identifier for the thread with thread identifier n (if the list defines a value for n, at all).

As for the method code generation, the transition from the sequential to the multi-threaded setting is rather straightforward. Concerning the methods' case switches (cf. Section 4.2), we merely have to replace the *next* expression by a $next[\texttt{tid}]$ expression. Thus, compared to Table 4.7 the case switch consists of conditional statements of the following form:

```
1 if((next[tid] == [ id ]) && [ check-where-clause ]) {
2   [ body ]
3   retVal = [ ret-val ];
4 } else {[ expectation_k ]};
```

However, additionally we have to consider the case that a thread enters the test program for the first time via an incoming method or constructor call. In this case, $next[\texttt{tid}]$ is not defined. Hence, we first have to check if the thread is new and, if so, we have to determine the matching thread specification for this thread. Note that only threads of externally defined thread classes may show up at the interface for the first time in terms of an incoming method or constructor call. Therefore, we can assume that the new thread is instantiated from an externally defined thread class and, correspondingly, only test thread specifications come into question for this matching procedure.

If a matching thread specification is found, then the *next* list is extended by `tid` such that *next*[`tid`] is initialized with the first expectation identifier of the matching thread specification. For a better understanding, consider an example specification which includes a test thread specification regarding thread class C_T that starts with an incoming method call expectation of method m of class C, i.e.,

```
test thread C_T{ [i](C x)?m(T y).where(e) { ... .
```

Moreover, assume that indeed an incoming call of method m of an instance of C via thread n has occurred, where n is new to the specification. In particular, *next*[n] is not defined. Then method m has to set *next*[n] to the expectation identifier i. Specifically, the above mentioned case switch in the body of method m has to be proceeded by the following code:

```
if (tid ∉ next) {
  if (tclass == C_T) {
    next[tid] = i
  } else {
    fail
  }
}
```

Thus, when `tid` is not in the *next* list, hence, when `tid` shows up for the first time, then method m checks the class type of the new thread by means of `tclass`. If the calling class is C_T then the *next* list is extended by `tid` that is mapped to the expectation identifier i. As we put this conditional statement at the very beginning of the method body, the above mentioned case switch can be executed subsequently.

So far, we have ignored two further problems that arise from the dynamic thread creation. First, we cannot resolve the local variable declaration problem with variable globalization, anymore (cf. Section 4.1.2). To understand this, consider the following test thread specification:

```
test thread C_T{
  [i](C x)?m1(T y).where(e) {
    o!m2() {
      (C x)?m1(T z).where(y = z) {
    ... },
```

The above specification consists of two nested incoming calls of $m1$ where the inner call's where-clause uses the parameter y of the outer call. As several instances of thread C_T may be created during the execution it is not sufficient to provide a (single) global pendant for y as we have done it in the sequential setting. Instead, for each thread we have to provide a corresponding set of global variables. Thus, similar to the solution for the global *next* variable, for each local variable we have to implement a dynamic list of globally accessible variables. Then, regarding the nested calls example given above, the second incoming call specification of $m1$ may access y by y[`tid`].

enter:

```
1 access = false;
2 while (!access) {
3   while(accID != 0) { };
4   accID = accID + tid;
5   if (addID == tid)
6     { access = true }
7   else
8     { accID = accID − tid }
9 }
```

exit:

```
1 access = false;
2 accID = accID − tid;
```

Table 7.5: *CoJapl* code generation: mutual exclusion

The second problem is due to the fact that a globally accessible list implementation must only allow a *mutually exclusive* writing access to the list, in order to avoid inconsistency. Therefore, in the following we provide a simple mutual exclusion algorithm for *CoJapl*. In particular, we assume that writing accesses to global lists are only realized within *critical sections*. Table 7.5 sketches entry and exit code to be executed by a thread whenever it wants to enter and, respectively, exit a critical section. The only assumption for this algorithm concerning the *CoJapl* language is that we consider thread identifiers to be represented by *integers* which can be added and subtracted. Each thread has a local variable *access* and additionally all threads share a global variable *accID*. The local variable *access* is used by a thread to indicate that is has access to the critical section. The global variable *addID* stores thread identifiers of competing threads. Let us have a closer look at the entry code. After initializing *access* to false, we enter the while-loop at Line 2. After that, we have to busy-wait for *accID* to become 0. A value of 0 indicates that no thread is in the critical section and that currently no thread has requested entrance to the critical section. A thread requests for entrance to the critical section by incrementing *accID* with its own thread it. If then afterwards *accID* indeed stores the thread identifier of the thread, then the thread is allowed to enter the section. Since other threads may have incremented the variable concurrently as well, however, *accID* may be unequal to the thread identifier. In this case, all competing threads have to decrement *accID* by their thread identifier again. Due to the fact that an assignment represents an *atomic computation step* in our language, there exist at most one thread which may find *accID* to store exactly its own thread identifier. Therefore, mutual exclusion is granted. When leaving the critical section, the thread again subtracts its identifier from *accID*.

Chapter 8

Concluding remarks

In this thesis, we presented a novel unit testing approach for object-oriented multi-purpose programming languages in the style of *Java* and C♯. Analyzing existing unit testing approaches we identified three goals to be melted into our testing framework:

- Due to the tendency that software developers do not only write the unit code but likewise are responsible for specifying and executing the corresponding unit test cases, the test specification language should be easily *accessible* by software developers. In particular, referring to the increasing popularity of agile software development methodologies, a specification language that is completely different to the programming language may hamper the propagated short test-and-develop cycles that many developers embark on.

- Object-orientation entails heavy collaboration among objects. As a consequence, within an object-oriented context, unit testing coincide with integration testing. Hence, an object-oriented unit test does not merely consist of relating input with output data but instead the proper *interaction of objects* itself is to be verified.

- Finally, not only due to the interdependency among objects, specifying interaction-based test cases can easily become quite complex. Therefore, it is useful if a test specification language allows to formalize test case specifications on a high level.

Our idea for combining these, partly contradicting, features was to define a test specification language by *extending* the programming language with dedicated *specification statements*. Aiming at interaction-based testing, the specification statements basically express expectations regarding the *observable behavior* of the unit under test.

Based on this, the main part of the thesis dealt with a unit testing framework for sequential object-oriented programs. The framework was introduced in four steps. Firstly, we presented a formally defined component-based object-oriented

programming language *Japl* that captured the basic features of *Java*, C♯, and similar languages. Second, we introduced the test specification language as an extension of *Japl*. As a third step, we developed a *test code generation algorithm* which allows to automatically generate a *Japl* test program from a test specification. A central contribution is the *correctness proof* of the code generation algorithm. We concluded the main part with a discussion about possible extensions of both, the programming language and the test specification language.

Even though the programming language *Japl*, as mentioned, captures only a small fraction of the features of today's commonly used object-oriented languages, its formal representation is already quite complex. In particular, the discussion about the language extension with subtyping and inheritance in Section 5.3 gives a foretaste of the complexity one would be confronted with, if aiming at a formal representation for the whole *Java* or C♯ language. One may conclude that implementations of these object-oriented languages are likewise so complex that compilers, virtual machines, and the like themselves should be thoroughly tested. But due to the lack of complete formal specifications regarding these languages, which would be essential to derive useful test cases, it is doubtful that these tests exist either.

Similarly, the development of the code generation algorithm and its correctness proof demonstrated the amount of considerations necessary for writing proper test code in general. Against this background, one specifically can get an impression on the effort that has to be made for writing interaction-based test code without the support of a tool that abstracts away some of the entailed intricacies.

The second part of the thesis introduced *concurrency* into the programming language by means of *thread classes*. Afterwards, we discussed a corresponding extension of the test specification language which also included a sketch regarding the necessary modifications of the code generation algorithm. Clearly, the above remarks about the difficulties of writing proper test code is even more true if concurrency comes into play [60].

As a conclusion, the testing framework proposed in this thesis, indeed, may facilitate writing interaction-based unit tests. Thus, regarding our testing approach, it suggests itself that implementing the framework in *Java* or C♯ is one of the next steps in the near future. The fact that parts of code generation algorithm is given in terms of simple functional programming language code may expedite the implementation. Moreover, as the specification language is defined as an extension of the programming language, it could be implemented with the help of an *extensible compiler* framework like *Polyglot* [51] or *The Dryad Compiler* [43], for instance. As indicated already, however, probably more effort will be necessary for embedding further features of real word programming languages into the formal framework. Besides subtyping and inheritance, synchronization mechanisms like synchronized methods and monitors represent interesting candidates for the

multi-threaded setting. Although it takes quite an effort to define a formal framework for interaction-based testing, it clearly has the benefit that it could form a basis for further testing-related research, in general. For instance, it could be useful in the field of *compositional testing* as it investigates how we can reuse test results about smaller components for the integration test of their composition [14].

Bibliography

[1] Abadi, M., Cardelli, L.: A Theory of Objects. Monographs in Computer Science. Springer-Verlag (1996)

[2] Ábrahám, E., de Boer, F.S., Bonsangue, M.M., Grüner, A., Steffen, M.: Observability, connectivity, and replay in a sequential calculus of classes. In: M. Bonsangue, F.S. de Boer, W.P. de Roever, S. Graf (eds.) Proceedings of the Third International Symposium on Formal Methods for Components and Objects (FMCO 2004), *Lecture Notes in Computer Science*, vol. 3657, pp. 296–316. Springer-Verlag (2005). URL `http://www.ifi.uio.no/~msteffen/download/fa-fmco.pdf`

[3] Ábrahám, E., Grabe, I., Grüner, A., Steffen, M.: Behavioral interface description of an object-oriented language with futures and promises. Journal of Logic and Algebraic Programming (2009). URL `http://www.ifi.uio.no/~msteffen/download/09/futures.pdf`. To appear in a special issue of the Journal of Logic and Algebraic Programming (NWPT'07).

[4] Ábrahám, E., Grüner, A., Steffen, M.: Abstract Interface Behavior of Object-Oriented Languages with Monitors. Theory of Computing Systems **43**(3-4), 322–361 (2008). DOI 10.1007/s00224-007-9047-0. URL `http://www.ifi.uio.no/~msteffen/download/07/monitors-journal.pdf`

[5] Ábrahám, E., Grüner, A., Steffen, M.: Heap-Abstraction for an Object-Oriented Calculus with Thread Classes. Journal of Software and Systems Modelling (SoSyM) **7**(2), 177–208 (2008). DOI 10.1007/s10270-007-0065-9. URL `http://springerlink.metapress.com/content/dqxhn70707442838/fulltext.pdf`

[6] Ammann, P., Offutt, J.: Introduction to Software Testing, 1 edn. Cambridge University Press (2008)

[7] Bandat, K.: On the Formal Definition of PL/I. In: AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, pp. 363–373. ACM, New York, NY, USA (1968). DOI http://doi.acm.org/10.1145/1468075.1468130

[8] Barbey, S., Buchs, D., Péraire, C.: A Theory of Specification-Based Testing for Object-Oriented Software. In: Proccedings of the European Dependable Computing Conference, *Lecture Notes in Computer Science*, vol. 1150. Springer-Verlag (1996)

[9] Baresi, L., Pezzè, M.: An Introduction to Software Testing. Electr. Notes Theor. Comput. Sci. **148**(1), 89–111 (2006)

[10] Beck, K.: Smalltalk Idioms: Simple Smalltalk Testing. The Smalltalk Report **4**(2) (1994). URL `http://www.xprogramming.com/testfram.htm`

[11] Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (2000)

[12] Beck, K., Gamma, E.: Test Infected: Programmers Love Writing Tests. Java Report **3**(7), 51–56 (1998)

[13] Beizer, B.: Software Testing Techniques, 2 edn. Van Nostrand Reinhold Co., New York, NY, USA (1990)

[14] Bertolino, A.: Software Testing Research: Achievements, Challenges, Dreams. In: Proceedings of Future of Software Engineering at ICSE 2007, pp. 85–103 (2007)

[15] Biermann, G., Parkinson, M.J., Pitts, A.M.: An Imperative Core Calculus for Java and Java with Effects. Technical Report 563, University of Cambridge Computer Laboratory (2004)

[16] Binder, R.V.: Testing Object-Oriented Systems, Models, Patterns, and Tools. Addison-Wesley (2000)

[17] Birrell, A.D.: An Introduction to Programming with C# Threads. Technical Report TR-2005-68, Microsoft Research Technical Report (2005)

[18] Boehm, B., Basili, V.R.: Software Defect Reduction Top 10 List. Computer **34**(1), 135–137 (2001). DOI http://doi.ieeecomputersociety.org/10.1109/2.962984

[19] Boehm, B.W.: Guidelines for Verifying and Validating Software Requirements and Design Specifications. In: Samet, P. A. (ed.):IFIP, pp. 711–719. North-Holland: Amsterdam (1979)

[20] de Boer, F.S., Bonsangue, M.M., Grüner, A., Steffen, M.: Java test driver generation from object-oriented interaction traces. In: Proceedings of the 2nd International Workshop on Harnessing Theories for Tool Support in Software TTSS'08, ICTAC 2008 satellite workshop, 30. August 2008, Istambul, Turkey (2008). URL `http://www.ifi.uio.no/~msteffen/download/08/javatestdriver.pdf`

[21] Byous, J.: Java Technology: The Early Years. Sun Developer Network (1998). URL `http://java.sun.com/features/1998/05/birthday.html`

[22] Dahl, O.J., Nygaard, K.: SIMULA — An ALGOL-Based Simulation Language. Communications of the ACM **9**(9), 671–678 (1966)

[23] Dijkstra, E.W.: Ewd 340: The Humble Programmer. Communications of the ACM **15** (1972). URL `http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD340.PDF`

[24] The easyMock home page (2007). URL `http://www.easymock.org`

[25] ECMA International Standardizing Information and Communication Systems: C# Language Specification, 4th edn. (2006). Standard ECMA-334

[26] The Eiffel home page (2008). URL `http://www.eiffel.com`

[27] Fowler, M.: Mocks Aren't Stubs (2007). URL `http://www.martinfowler.com/articles/mocksArentStubs.html`

[28] Freeman, S., Pruyce, N., Mackinnon, T., Walmes, J.: Mock Roles, not Objects. In: Ninetheeth Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '04. ACM (2004). In *SIGPLAN Notices*

[29] Freeman, S., Pryce, N.: Evolving an Embedded Domain-Specific Language in Java. In: P.L. Tarr, W.R. Cook (eds.) OOPSLA Companion, pp. "855–865". ACM (2006)

[30] Goldberg, A., Robson, D.: Smalltalk-80: The Language and its Implemementation. Addison-Wesley, Reading, MA (1983)

[31] Gordon, A.D., Hankin, P.D.: A Concurrent Object Calculus: Reduction and Typing. In: U. Nestmann, B.C. Pierce (eds.) Proceedings of HLCL '98, *Electronic Notes in Theoretical Computer Science*, vol. 16.3. Elsevier Science Publishers (1998). URL `http://www.elsevier.nl/locate/entcs/volume16.3.html`

[32] Guttag, J.V., Horning, J.J., Wing, J.M.: Larch in Five Easy Pieces. Tech. Rep. 5, DEC System Research Center, 130 Lytton Avenue, Palo Alto, CA 94301 (1985). Order from src-report@src.dec.com

[33] Harrold, M.J.: Testing: A roadmap. In: In The Future of Software Engineering, pp. 61–72. ACM Press (2000)

[34] Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. In: Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '99, pp. 132–146. ACM (1999). In *SIGPLAN Notices*

[35] Jacky, J., Veanes, M., Campbell, C., Schulte, W.: Model-Based Software Testing and Analysis with C$^\sharp$. Cambridge University Press (2008)

[36] JSR 175: A Metadata Facility for the Java Programming Language (2004). URL `http://jcp.org/en/jsr/detail?id=175`

[37] The Java Modeling Language (JML) home page (2003). URL `http://www.cs.iastate.edu/~leavens/JML/`

[38] The jMock home page (2007). URL `http://www.jmock.org`

[39] Jones, C.: Applied Software Management: Assuring Productivity and Quality. McGraw-Hill (1996)

[40] Jones, C.B.: Systematic Software Development Using VDM, 2 edn. International Series in Computer Science. Prentice Hall (1990)

[41] The JUnit home page (2007). URL `http://junit.sourceforge.net`

[42] Kaner, C., Nguyen, H.Q., Falk, J.L.: Testing Computer Software. John Wiley & Sons, Inc., New York, NY, USA (1993)

[43] Kats, L.C.L., Bravenboer, M., Visser, E.: Mixing Source and Bytecode. A Case for Compilation by Normalization. In: G. Kiczales (ed.) Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008), pp. 91–108. ACM, New York, NY, USA (2008)

[44] Leavens, G.T., Cheon, Y.: Design by Contract with JML (2006). URL `http://www.jmlspecs.org/jmldbc.pdf`

[45] Mackinnon, T., Freeman, S., Craig, P.: Endo-Testing: Unit Testing with Mock Objects. In: G. Succi, M. Marchesi (eds.) Extreme Programming Examined, The XP Series, pp. 287–301. Addison-Wesley (2001)

[46] Meyer, B.: Applying "Design by Contract". IEEE Computer **25**(10), 40–51 (1992)

[47] Milner, R.: An algebraic definition of simulation between programs. In: Proceedings of 2nd Joint Conference on Artificial Intelligence, pp. 481–489. BCS (1971)

[48] Milner, R.: Communication and Concurrency. Prentice Hall (1989)

[49] Morris, J.H.: Lambda Calculus Models of Programming Languages. Technical Report MIT-LCS TR-57, MIT Press (1968)

[50] Myers, G.J.: The Art of Software Testing, 2nd edn. Wiley (2004)

[51] Nathaniel Nystrom Michael R. Clarkson, A.C.M.: Polyglot: An Extensible Compiler Framework for Java. In: Proceedings of the 12th International Conference on Compiler Construction. Springer (2003)

[52] Naur, P., Randell, B. (eds.): Software Engineering: A Report on a Conference sponsored by the NATO science committee. NATO (1969). URL `http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF`

[53] de Nicola, R., Hennessy, M.: Testing Equivalences for Processes. Theoretical Computer Science **34**, 83–133 (1984)

[54] Park, D.M.R.: Concurrency and automata on infinite sequences. In: P. Deussen (ed.) Fifth GI Conference on Theoretical Computer Science, *Lecture Notes in Computer Science*, vol. 104, pp. 167–183. Springer-Verlag (1981)

[55] Patton, R.: Software Testing, 2nd edn. SAMS (2005)

[56] Plotkin, G.D.: A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University (1981)

[57] Prasetya, W., Vos, T., Baars, A.: Trace-based Reflexive Testing of OO Programs with T2. In: ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation, pp. 151–160. IEEE Computer Society, Washington, DC, USA (2008). DOI http://dx.doi.org/10.1109/ICST.2008.12

[58] Rahn, C.: Bloomberg.com (2009). URL `http://www.bloomberg.com/apps/news?pid=20601100&sid=avUsNKymzMPk`

[59] Reynolds, J.C.: Theories of Programming Languages. Cambridge University Press (1998)

[60] Roscoe, A.W.: Theory and Practice of Concurrency. Prentice Hall (1998)

[61] Spillner, A., Linz, T., Schaefer, H.: Software Testing Foundations: A Study Guide for the Certified Tester Exam, 1 edn. Rocky Nook (2006)

[62] Std.1008-1987, I.: IEEE Standard for Software Unit Testing. The Institute of Electrical and Electronics Engineers: New York (1993)

[63] Std.610.121990, I.: IEEE Standard Glossary of Software Engineering Terminology. The Institute of Electrical and Electronics Engineers: New York (1990)

[64] Steffen, M.: Object-Connectivity and Observability for Class-Based, Object-Oriented Languages. Habilitation thesis, Technische Faktultät der Christian-Albrechts-Universität zu Kiel (2006)

[65] Stevens, P., Pooley, R.: Using UML: Software engineering with Objects and Components. Object Technology Series. Addison-Wesley Longman (1999)

[66] Strniša, R., Sewell, P., Parkinson, M.: The Java Module System: Core Design and Semantic Definition. In: OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications, pp. 499–514. ACM, New York, NY, USA (2007). DOI 10.1145/1297027.1297064

[67] Stroustrup, B.: The C++ Programming Language. Addison-Wesley (1986)

[68] The TestNG home page (2008). URL `http://java-source.net/open-source/testing-tools/testng`

[69] Tretmans, G.J.: A formal approach to conformance testing. Ph.D. thesis, Enschede (1992). URL `http://doc.utwente.nl/58114/`

[70] Methods for Testing and Specification (MTS). The Testing and Test Control Notation Version 3 (TTCN-3)". European Standard ETSI ES 201 8731 v2.2.1 (2002)

[71] Tucker, A.B. (ed.): The Computer Science and Engineering Handbook. CRC Press, Inc., Boca Raton, FL, USA (2004)

[72] Wright, G.R., Stevens, W.R.: TCP/IP Illustrated: The implementation. Addison-Wesley (2004)

Part III

Proofs

Appendices

Appendix A

Subject reduction

This chapter deals with the with well-typedness of configuration. We want to prove that the rules of the operational semantics preserve well-typedness of the configuration. This feature, called *subject reduction*, was formalized in Lemma 2.4.7 and what follows is the proof for this lemma. Definition 2.4.4 introduces three requirements for well-typed configurations and the idea of the proof is to make a case analysis on the transition for each requirement.

Proof. By case analysis of the transition step. As a precondition for all cases, we assume that $\Delta \vdash c : \Theta$ holds. Let h and h' be the heap functions as well as v and v' the global variable functions for the configuration c and c', respectively. Before we start with the case analysis, let us make three general observations. First, no transition rule changes the domain of the global variable function, i.e. $dom(\mathsf{v}) = dom(\mathsf{v}')$. Second, regarding external steps the new assumption-commitment context always represents an extension of the previous context. In particular, all class names in Δ and in Θ have the same type in Δ' and in Θ', respectively. Furthermore, all transition steps change the local variables and code of the top-most activation records only, if at all. Thus, within the following proof we can ignore the tail of the call stack and focus on the top-most activation records.

Now let us prove the first requirement of Definition 2.4.4, i.e., we want to show that all objects on the heap of configuration c' belong to a program class mentioned in Θ'.

Case Let us assume that $c \rightsquigarrow_p c'$.
Regarding the Rules Ass, Call, BlkBeg, BlkEnd, While$_i$, Cond$_i$, and Ret there is no change of the heap involved. As Θ' is an extension of Θ compliance with the first requirement results from the precondition.

Subcase Rule FUpd
Lets assume that $c \rightsquigarrow c'$ due to a field update. In particular, the third premise of Rule FUpd implements the actual update. It also shows, however, that the class name of the involved object is not changed. Thus, a field update does not break the requirement.

Subcase Rule NEW
Assume that c evolves to c' due to application of Rule NEW. Then the heap is extended by a new object o of class C. Likewise, the stack is extended by the method body of C. Since the auxiliary function *cbody* is only defined for program classes and as the program p is well-typed, we can deduce that $\Theta' \vdash C : [\![\ldots]\!]$.

Case Let us now assume that $\Delta \vdash c : \Theta \xrightarrow{a}_p \Delta' \vdash c' : \Theta'$.
Only one rule of the external semantics changes the heap, namely Rule NEWI. Since Θ' is an extension of Θ the requirement follows from the precondition for all the other external rules. Regarding NEWI, as in Rule NEW, we can basically deduce from the definedness of *cbody* for class name C that the first requirement of a well-typed configuration also holds for the new configuration with the extended heap. Now let us prove the second requirement of Definition 2.4.4. That is, we have to show that every free variable of each activation record of c' is a global variable or in the domain of the record's local variable list.

Case Again consider $c \rightsquigarrow_p c'$
We show the most interesting cases.

Subcase Rule ASS
Execution of the assignment statement $x = e$ does not extend the set of free variables of the corresponding activation record but instead possibly reduces it by x and $fvars(e)$. Moreover, the domain of the record's local variable list is not changed which yields the proof for the requirement.

Subcase Rule CALL and Rule NEW
Transitions that represent an internal method call or object instantiation create a new top most activation record, while the method or constructor call in the previously top most record is replaced by a receive statement. Thus, regarding the previously top most record, all free variables of the record's code are part of the record's local variable list. As for the new activation record, the code is instantiated by the method or constructor body of the corresponding program class. We know that the program is well-typed, therefore the code might only make references to global variables, to `this`, or to local variables of the method itself. Since the new record is equipped with a local variable function that consists of a mapping for the aforementioned variables, the requirement is fulfilled.

Subcase Rule RET
An application of Rule RET causes the removal of the top most activation record. Apart from this, only the receive statement on top of the calling activation record is removed. Thus, again all free variables of the new top most activation record are in the record's local variable list.

Case Assume $\Delta \vdash c : \Theta \xrightarrow{a}_p \Delta' \vdash c' : \Theta'$

Subcase RulesCALLO and NEWO
In both cases the outgoing method or constructor call is replaced by an annotated receive statement. No introduction of new variables and no modification of the

record's local variable functions is involved in this step. Thus the requirement follows from the precondition.

Subcase Rule RetO
Only the top most activation record is removed. The requirement follows from the precondition.

Subcase Rules CallI and NewI
Both rules extend the call stack by a new activation record leaving the rest of the call stack unchanged. Like in the case for internal method calls we can deduce from the well-typedness of the program that the new activation record conforms to the second requirement of the well-typedness definition for configurations.

Subcase Rule RetI
An incoming return leads to the removal of the receive statement on top of the top most activation record. Again, no new free variables are introduced and the domain of the local variable function list is not changed. Finally, we have to prove that also the third requirement for well-typed configurations is fulfilled by the new configuration c'. More specifically, we have to show that each of the call stack's activation records that represents a method or constructor execution provides a valid value for the special name `this`. Obviously, the only interesting cases are the transitions that deal with internal or incoming method and constructor calls. All other transitions do not modify the value of `this` within the local variable lists.

Case Internal step

Subcase Rule Call
The local variable function for the new activation record maps `this` to o. Moreover, the second premise of the rule verifies that o indeed is on the heap.

Subcase Rule New
In Rule New also `this` is mapped to o. In the object creation case, however, the object o is created and the new heap is extended by the new object.

Case External step

Subcase Rule CallI
The argumentation for the incoming method call is almost identical to the proof for internal method calls. The first premise of the label check T-CallI verifies that the callee object name o represents an object that is committed by the program. Furthermore, the local variable function of the new activation record maps `this` to o.

Subcase Rule NewI
Similar to the internal object creation, we can see in Rule NewI that the heap is extended with a new object referenced by o which in turn serves as the value for `this` in the local variable function. □

Appendix B

Compositionality

The goal of this section is to prove the compositionality-Lemma 2.5.5 of Section 2.5. This is structured as follows. We start with the discussion of some general features of the language's transition semantics. Afterwards we will provide a merge definition that meets the requirements of the merge function definition given in Lemma 2.5.5. This is followed by a few small proofs of some simple yet useful features of the merge function in general. The compositionality-Lemma states that the order regarding the application of the merge function on configurations, on the one hand, and application of the transition rules, on the other hand, does not play a role. Thus, the lemma consists of two directions: one direction states that regarding the transition semantics the composition of two components evolves to the same result as the two original components. The other direction says that two constituents of one (closed) program evolve to the same result as the original program. Correspondingly, the proof of Lemma 2.5.5 actually consists of two parts. First, we will show certain features about the composition of two components. Then, we show the features about the constituents of a closed program. Both cases, however, consist of several smaller sub-proofs, but the schema for both parts is the same. That is, regarding the composition we first prove the features for single internal and single external steps. Then the compositionality part follows from this by induction on the length of the trace. Similarly, regarding the decomposition we show that a single internal step of a closed program corresponds to internal or external single steps with regards to its constituents. Again, the decompositionality direction follows by induction on the length of the trace.

We begin with three small lemmas about the independence of internal deductions from certain changes regarding the stack, heap, global variables, or the component code. More specifically, the first lemma states that a single internal deduction step does only depend on the topmost but not on the trailing activation records of the call stack.

Lemma B.0.1 (Stack tail does not influence internal steps)**:** Assume two configurations

$$(h, \mathsf{v}, \mathsf{CS} \circ \mathsf{CS}_1^b), (h, \mathsf{v}, \mathsf{CS} \circ \mathsf{CS}_2^b) \in \mathit{Conf}.$$

If $(h, \mathsf{v}, \mathsf{CS} \circ \mathsf{CS}_1^b) \rightsquigarrow (h', \mathsf{v}', \acute{\mathsf{CS}} \circ \mathsf{CS}_1^b)$ then also $(h, \mathsf{v}, \mathsf{CS} \circ \mathsf{CS}_2^b) \rightsquigarrow (h', \mathsf{v}', \acute{\mathsf{CS}} \circ \mathsf{CS}_2^b)$.

Proof. By case analysis on the computation step. As for simple computation steps, i.e., computation steps which do only modify the top most activation record, the lemma follows immediately from the corresponding rules of the internal operational semantics, which are Ass, FUpd, BlkBeg, BlkEnd, Whl$_i$, and Cond$_i$. The remaining internal rules, Call, New, and Ret, deserve a closer look, as they also change the number of activation records within the call stack.

Case Rule Call
In case of an internal method call we can assume that

$$\mathsf{CS} = (\mu, x = e.m(\overline{e}); mc)$$

and correspondingly that

$$\acute{\mathsf{CS}} = (\mathsf{v}_l, mbody(C, m)) \circ (\mu, \mathtt{rcv}x; mc) \ .$$

Now it is easy to see that the application of Rule Call is independent of the call stack tail CS_1^b and CS_2^b, respectively.

Case Rule New
Similar to internal method calls, regarding internal constructor calls we can assume that

$$\mathsf{CS} = (\mu, x = \mathtt{new}\ C(\overline{e}); mc)$$

and correspondingly that

$$\acute{\mathsf{CS}} = (\mathsf{v}_l, cbody(C)) \circ (\mu, \mathtt{rcv}x; mc) \ .$$

Again, Rule New is formulated independently of the call stack tail CS_1^b and CS_2^b, respectively.

Case Rule Ret
As for an internal method or constructor return, we can define

$$\mathsf{CS} = (\mu_1, \mathtt{return}\ e) \circ (\mu_2, \mathtt{rcv}\ x; mc)$$

and

$$\acute{\mathsf{CS}} = (\mu_2', mc) \ .$$

Yet again, this definition makes the independence of Rule Ret regarding the call stack tail apparent. □

Similarly, extensions of the heap or of the global variable function do not influence the outcome of internal computation steps. This is formalized in the next lemma. For two functions f_1 and f_2 with $dom(f_1) \perp dom(f_2)$ we use the notion $f_1 \frown f_2$ for the function that represents the disjunct union of f_1 and f_2.

Lemma B.0.2 (Heap and variable extension do not affect internal steps)**:** If $(h_1, \mathsf{v}_1, \mathsf{CS}) \rightsquigarrow (h_1', \mathsf{v}_1', \mathsf{CS}')$ such that $dom(h_1') \perp dom(h_2)$ then also

$$(h_1 \frown h_2, \mathsf{v}_1 \frown \mathsf{v}_2, \mathsf{CS}) \rightsquigarrow (h_1' \frown h_2, \mathsf{v}_1' \frown \mathsf{v}_2, \mathsf{CS}') \ .$$

Proof. Applicability of the internal transition $(h, \mathsf{v}, \mathsf{CS}) \rightsquigarrow (h', \mathsf{v}', \mathsf{CS}')$ ensures that the deduction step does not realize a call to an external class or object and that only evaluation of local variables defined in CS, of global variables of v, or object names of h might be involved. Disjunction of h_1' and h_2 is required in order to prevent name clashes due to internal object creation. This, however, does not represent a real restriction, since we consider the semantics modulo renaming anyway, as we have remarked in 2.4.6 already. □

Also extending the program by another component does not affect the outcome of an internal step.

Lemma B.0.3 (Additional classes do not affect internal steps)**:** Assume two components p and p' such that $p \uplus p'$ is defined. If $(h, \mathsf{v}, \mathsf{CS}) \rightsquigarrow_p (h', \mathsf{v}', \mathsf{CS}')$ then also $(h, \mathsf{v}, \mathsf{CS}) \rightsquigarrow_{p \uplus p'} (h', \mathsf{v}', \mathsf{CS}')$.

Proof. Trivial, as the reduction step does only refer to method code of p, if at all. And the component merge does not modify method code of p. □

Now its time to give a concrete definition of a merge function. This merge function will form the basis of the compositionality proof.

Definition B.0.4 (Merge of configurations)**:** Given two configurations

$$(h_1, \mathsf{v}_1, \mathsf{CS}_1), (h_2, \mathsf{v}_2, \mathsf{CS}_2) \in \mathit{Conf}$$

with $\Delta \vdash (h_1, \mathsf{v}_1, \mathsf{CS}_1) : \Theta$ and $\Theta \vdash (h_2, \mathsf{v}_2, \mathsf{CS}_2) : \Delta$. We assume that $dom(h_1) \perp dom(h_2)$ as well as $dom(\mathsf{v}_1) \perp dom(\mathsf{v}_2)$ – otherwise we assume a proper renaming of objects or, respectively, variables. The result of the merge

$$(h, \mathsf{v}, \mathsf{CS}) = (h_1, \mathsf{v}_1, \mathsf{CS}_1) \uplus (h_2, \mathsf{v}_2, \mathsf{CS}_2)$$

is defined by:

- $h \stackrel{\text{def}}{=} h_1 \frown h_2$,
- $v \stackrel{\text{def}}{=} v_1 \frown v_2$, and
- $\mathsf{CS} \stackrel{\text{def}}{=} \mathsf{CS}_1 \mathbin{\wedge\!\!\!\wedge} \mathsf{CS}_2$, where $\mathbin{\wedge\!\!\!\wedge}$ denotes a commutative operation representing the merge of the two call stacks which is inductively defined by the following equations:

$$(\mathsf{AR}^i \circ \mathsf{AR}^{ib} \circ \mathsf{CS}_1^b) \mathbin{\wedge\!\!\!\wedge} \mathsf{CS}_2^{eb} \stackrel{\text{def}}{=} \mathsf{AR}^i \circ (\mathsf{AR}^{ib} \circ \mathsf{CS}_1^b) \mathbin{\wedge\!\!\!\wedge} \mathsf{CS}_2^{eb} \tag{B.1}$$

$$(\mathsf{AR}^i \circ \mathsf{CS}_1^{eb}) \mathbin{\wedge\!\!\!\wedge} (\mathsf{AR}_2^{eb} \circ \mathsf{CS}_2^b) \stackrel{\text{def}}{=} \mathsf{AR}^i \circ \mathsf{CS}_1^{eb} \mathbin{\wedge\!\!\!\wedge} (\mathsf{AR}_2^{ib} \circ \mathsf{CS}_2^b) \tag{B.2}$$

$$\mathsf{AR}^i \mathbin{\wedge\!\!\!\wedge} (\mathsf{AR}_2^{eb} \circ \mathsf{CS}_2^b) \stackrel{\text{def}}{=} \mathsf{AR}^i \circ (\mathsf{AR}_2^{ib} \circ \mathsf{CS}_2^b) \tag{B.3}$$

$$\mathsf{AR}^i \circ \mathsf{CS}_1^b \mathbin{\wedge\!\!\!\wedge} \epsilon \stackrel{\text{def}}{=} \mathsf{AR}^i \circ \mathsf{CS}_1^b \tag{B.4}$$

Note that in AR_2^{ib} denotes the activation record that results from AR_2^{eb} by forgetting the return type of the topmost `rcv` statement.

Remark B.0.5: The equations in Definition B.0.4 show that a merge of two call stacks is only defined if exactly one call stack has an active or internally blocked activation record on top and the other call stack is externally blocked.

The next lemma makes a statement about the merge of call stacks.

Lemma B.0.6 (Topmost activation record remains topmost)**:** There exists a function f such that for all defined merges of call stacks the following holds:

1. $(\mathsf{AR}^i \circ \mathsf{CS}_1^b) \mathbin{/\!\!\backslash\!\!\backslash} \mathsf{CS}_2^b = \mathsf{AR}^i \circ f(\mathsf{CS}_1^b, \mathsf{CS}_2^b)$.

2. In particular, the activation record that is on top of the active call stack before the merge also remains the topmost record of the resulting call stack after the merge. Moreover, the form of the rest of the resulting call stack does not depend on the topmost record but is determined only by the rest of the first stack frame and the second stack frame.

Proof. Let the function f be defined by

$$f(\mathsf{CS}_1, \mathsf{CS}_2) \stackrel{\text{def}}{=} \begin{cases} (\mathsf{AR}_1^{eb} \circ \mathsf{CS}_1^b) \mathbin{/\!\!\backslash\!\!\backslash} (\mathsf{AR}_2^{ib} \circ \mathsf{CS}_2^b) & \text{if } \mathsf{CS}_1 = \mathsf{AR}_1^{eb} \circ \mathsf{CS}_1^b \text{ and} \\ & \quad \mathsf{CS}_2 = \mathsf{AR}_2^{eb} \circ \mathsf{CS}_2^b \\ \mathsf{CS}_1 \mathbin{/\!\!\backslash\!\!\backslash} \mathsf{CS}_2 & \text{else} \end{cases}$$

where AR_2^{ib} represents the activation record which results from AR_2^{eb} by forgetting the type annotation of the receive statement. Then f has the property stated in the first statement. The second statement follows immediately from the definition of the merge of two stack frames. □

Now we want to apply the new lemmas in order to show that a simple internal computation step of one configuration will not be influenced if we merge it with another configuration. This is formalized in the following lemma.

Lemma B.0.7 (Merge does not influence simple deduction)**:** Assume a configuration $(h_1, \mathsf{v}_1, \mathsf{AR}^a \circ \mathsf{CS}^b)$ such that

$$(h_1, \mathsf{v}_1, \mathsf{AR}^a \circ \mathsf{CS}^b) \rightsquigarrow (h_1', \mathsf{v}_1', \acute{\mathsf{AR}}^a \circ \mathsf{CS}^b)$$

represents a simple deduction. Then, if for some other configuration $(h_2, \mathsf{v}_2, \mathsf{CS}_2^b)$ the merge $(h_1, \mathsf{v}_1, \mathsf{AR}^a \circ \mathsf{CS}^b) \mathbin{\ni\!\!\!

We know from Lemma B.0.6 that $\mathsf{AR}^a \circ \mathsf{CS}^b \mathbin{/\!\!\backslash\!\!\backslash} \mathsf{CS}^b_2 = \mathsf{AR}^a \circ f(\mathsf{CS}^b, \mathsf{CS}^b_2)$. From Lemma B.0.1 and Lemma B.0.2 we can deduce

$$(h_1 \frown h_2, \mathsf{v}_1 \frown \mathsf{v}_2, \mathsf{AR}^a \circ f(\mathsf{CS}^b, \mathsf{CS}^b_2)) \rightsquigarrow$$
$$(h'_1 \frown h_2, \mathsf{v}'_1 \frown \mathsf{v}_2, \acute{\mathsf{AR}}^a \circ f(\mathsf{CS}^b, \mathsf{CS}^b_2)) = (h'_1, \mathsf{v}'_1, \acute{\mathsf{AR}}^a \circ \mathsf{CS}^b) \Join (h_2, \mathsf{v}_2, \mathsf{CS}^b_2).$$

□

Note that we didn't index the transition arrow in the previous lemma, as the lemma is independent of a certain program code. However, we certainly assume that all transitions in the lemma are understood in the context of the same program.

The next two lemmas will show one of the compositionality properties for single steps of the operational semantics. More specifically, Lemma B.0.8 states that for internal computation steps the order regarding merge operation application and transition rule application does not matter. Afterwards Lemma B.0.9 will show the same property for external computation steps.

Lemma B.0.8 ($\Join$ and $\rightsquigarrow$)**:** For two configurations $c_1, c_2 \in \mathit{Conf}$ and two component p_1 and p_2 such that $c_1 \Join c_2$ and $p_1 \Join p_2$ is defined, the following holds: If $c_1 \rightsquigarrow_{p_1} c'_1$ then $c_1 \Join c_2 \rightsquigarrow_{p_1 \Join p_2} c'_1 \Join c_2$.

Proof. For simple computation steps the property has been proven by Lemma B.0.7 already. It remains to show the property also for the other internal transition rules given in Table 2.7. Let $c_1 = (h_1, \mathsf{v}_1, (\mathsf{AR}^a \circ \mathsf{CS}^b_1))$ and $c_2 = (h_2, \mathsf{v}_2, \mathsf{CS}^b_2)$.

Case Rule Ret
Applicability of Rule Ret for c_1 implies

$$c_1 = (h_1, \mathsf{v}_1, (\mu, \texttt{return}\ e) \circ (\mu', \texttt{rcv}\ x;\ mc) \circ \mathsf{CS}^b) \rightsquigarrow (h_1, \mathsf{v}'_1, (\mu'', mc) \circ \mathsf{CS}^b).$$

Moreover, applying Equation B.1 twice as well as rule Ret, Lemma B.0.2, and Lemma B.0.1 yields

$$c_1 \Join c_2 = (h_1 \frown h_2, \mathsf{v}_1 \frown \mathsf{v}_2, (\mu, \texttt{return}\ e) \circ (\mu', \texttt{rcv}\ x;\ mc) \circ (\mathsf{CS}^b \mathbin{/\!\!\backslash\!\!\backslash} \mathsf{CS}^b_2)) \rightsquigarrow$$
$$(h_1 \frown h_2, \mathsf{v}'_1 \frown \mathsf{v}_2, (\mu'', mc) \circ (\mathsf{CS}^b \mathbin{/\!\!\backslash\!\!\backslash} \mathsf{CS}^b_2)).$$

On the other hand Equation B.1 yields

$$(h_1, \mathsf{v}'_1, (\mu'', mc) \circ \mathsf{CS}^b) \Join c_2 = (h_1 \frown h_2, \mathsf{v}'_1 \frown \mathsf{v}_2, (\mu'', mc) \circ (\mathsf{CS}^b \mathbin{/\!\!\backslash\!\!\backslash} \mathsf{CS}^b_2)).$$

Case Rule Call
Applicability of Rule Ret for c_1 implies

$$c_1 = (h_1, \mathsf{v}_1, (\mu, x = e.m(\overline{e}); mc) \circ \mathsf{CS}^b_1) \rightsquigarrow (h_1, \mathsf{v}_1, \mathsf{AR}^a_m \circ (\mu, \texttt{rcv}\ x; mc) \circ \mathsf{CS}^b_1),$$

where AR^a_m represents the activation record that comprises the method body of the called method m. Again, by applying Equation B.1, rule CALL, Lemma B.0.2, and Lemma B.0.1 we get

$$c_1 \Join c_2 = (h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2, (\mu, x = e.m(\overline{e}); mc) \circ (\mathsf{CS}^b_1 \mathbin{/\!\!\backslash\!\!\backslash} \mathsf{CS}^b_2) \rightsquigarrow$$
$$(h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2, \mathsf{AR}^a_m \circ (\mu, \texttt{rcv}\ x; mc) \circ (\mathsf{CS}^b_1 \mathbin{/\!\!\backslash\!\!\backslash} \mathsf{CS}^b_2).$$

On the other hand, applying Equation B.1 twice yields

$$(h_1, \mathsf{v}_1, \mathsf{AR}^a_m \circ (\mu, \texttt{rcv}\ x; mc) \circ \mathsf{CS}^b_1) \Join c_2 =$$
$$(h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2, \mathsf{AR}^a_m \circ (\mu, \texttt{rcv}\ x; mc) \circ (\mathsf{CS}^b_1 \mathbin{/\!\!\backslash\!\!\backslash} \mathsf{CS}^b_2).$$

Case Rule NEW
The proof is almost identical to the proof for method calls. □

Lemma B.0.9 ($\Join$ and $\xrightarrow{a}$)**:** Assume two components p_1 and p_2 as well as configurations $c_1, c_2 \in \mathit{Conf}$ such that $p_1 \Join p_2$ and $c = c_1 \Join c_2$ are defined. Further, assume $\Delta \vdash c_1 : \Theta \xrightarrow{a}_{p_1} \Delta' \vdash c_1' : \Theta'$ as well as $\Theta \vdash c_2 : \Delta \xrightarrow{\bar{a}}_{p_2} \Theta' \vdash c_2' : \Delta'$. Then $c_1 \Join c_2 \rightsquigarrow_{p_1 \Join p_2} c_1' \Join c_2'$ as well as $c_1 \Join c_2 \rightsquigarrow_{p_2 \Join p_1} c_1' \Join c_2'$.

Proof. **Case** $a = \nu(\Theta').\langle \mathit{call}\ o.m(\overline{v})\rangle!$
In this case we know from rule CALLO that

$$c_1 = (h_1, \mathsf{v}_1, (\mu, x = e.m(\overline{e}); mc) \circ \mathsf{CS}^b)$$

such that $[\![e]\!]^{\mathsf{v}_1,\mu}_{h_1} = o$ and $[\![\overline{e}]\!]^{\mathsf{v}_1,\mu}_{h_1} = \overline{v}$. Moreover the rule yields

$$c_1' = (h_1, \mathsf{v}_1, (\mu, \texttt{rcv}\ x{:}T; mc) \circ \mathsf{CS}^b)$$

On the other hand, from rule CALLI and from the complementary label $\bar{a}$ we can deduce for c_2 that

$$c_2 = (h_2, \mathsf{v}_2, \mathsf{CS}^{eb}_2) \text{ and } c_2' = (h_2, \mathsf{v}_2, \mathsf{AR}^a_m \circ \mathsf{CS}^{eb}_2).$$

It is $[\![e]\!]^{\mathsf{v}_1{}^\frown \mathsf{v}_2,\mu}_{h_1{}^\frown h_2} = [\![e]\!]^{\mathsf{v}_1,\mu}_{h_1}$ as well as $[\![\overline{e}]\!]^{\mathsf{v}_1{}^\frown \mathsf{v}_2,\mu}_{h_1{}^\frown h_2} = [\![\overline{e}]\!]^{\mathsf{v}_1,\mu}_{h_1}$. Thus, Lemma B.0.6 and Rule CALL yield

$$c_1 \Join c_2 = (h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2, (\mu, x = e.m(\overline{e}); mc) \circ f(\mathsf{CS}^b, \mathsf{CS}^{eb}_2)) \rightsquigarrow_{p_1 \Join p_2}$$
$$(h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2, \mathsf{AR}^a_m \circ (\mu, \texttt{rcv}\ x; mc) \circ f(\mathsf{CS}^b, \mathsf{CS}^{eb}_2)).$$

Finally, due to Equation B.0.4 and Lemma B.0.6 we get

$$\begin{aligned} c_1' \Join c_2' &= (h_1, \mathsf{v}_1, (\mu, \texttt{rcv}\ x{:}T; mc) \circ \mathsf{CS}^b) \Join (h_2, \mathsf{v}_2, \mathsf{AR}^a_m \circ \mathsf{CS}^{eb}_2) \\ &= (h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2, \mathsf{AR}^a_m \circ f(\mathsf{CS}^{eb}_2, (\mu, \texttt{rcv}\ x{:}T; mc) \circ \mathsf{CS}^b)) \\ &= (h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2, \mathsf{AR}^a_m \circ (\mu, \texttt{rcv}\ x; mc) \circ f(\mathsf{CS}^b, \mathsf{CS}^{eb}_2)). \end{aligned}$$

$\boxed{\textbf{Case}}$ $a = \nu(\Theta').\langle return(v)\rangle!$
According to rule RETO it is

$$c_1 = (h_1, \mathsf{v}_1, (\mu_1, \texttt{return}\ e) \circ \mathsf{CS}_1^{eb}) \ \text{ such that } \ [\![e]\!]_{h_1}^{\mathsf{v}_1,\mu_1} = v$$

and $c_1' = (h_1, \mathsf{v}_1, \mathsf{CS}_1^{eb})$. Likewise we know from rule RETI that

$$c_2 = (h_2, \mathsf{v}_2, (\mu_2, \texttt{rcv}\ x{:}T; mc) \circ \mathsf{CS}_2^b) \ \text{ and } \ c_2' = (h_2, \mathsf{v}_2', (\mu_2', mc) \circ \mathsf{CS}_2^b).$$

Now, due to Equation B.1, Equation B.0.4, Lemma B.0.6, and Lemma B.0.2 we get

$$\begin{aligned} c_1 \Supset c_2 &= (h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2, (\mu_1, \texttt{return}\ e) \circ (\mathsf{CS}_1^{eb} \mathbin{/\!\!\backslash\!\!\backslash} (\mu_2, \texttt{rcv}\ x{:}T; mc) \circ \mathsf{CS}_2^b) \\ &= (h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2, (\mu_1, \texttt{return}\ e) \circ (\mu_2, \texttt{rcv}\ x; mc) \circ f(\mathsf{CS}_2^b, \mathsf{CS}_1^{eb}) \rightsquigarrow \\ &\quad (h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2', (\mu_2', mc) \circ f(\mathsf{CS}_2^b, \mathsf{CS}_1^{eb}) \end{aligned}$$

On the other hand, Lemma B.0.6 yields

$$\begin{aligned} c_1' \Supset c_2' &= (h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2', \mathsf{CS}_1^{eb} \mathbin{/\!\!\backslash\!\!\backslash} (\mu_2', mc) \circ \mathsf{CS}_2^b) \\ &= (h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2', (\mu_2', mc) \circ f(\mathsf{CS}_2^b, \mathsf{CS}_1^{eb})). \end{aligned}$$

All other cases are similar or dual. □

In the following we want to prove the other implication of the compositionality lemma. That is, we want to show that a component's sub-constituents come to the same result as the original component. However, again we first start by introducing some auxiliary lemmas. In particular the next lemma states that regarding an internal computation step one can prune the heap and the global variable function of a configuration to a minimum without influencing the outcome of the computation. More specifically, in most cases the heap can be even reduced to the object that is referenced by the variable `this` of the topmost activation record, as only field updates or field lookups of the corresponding object might be involved in the computation step. An exception is a method invocation where we also have to include the callee object into the minimal heap.

Lemma B.0.10 (Reduction of heap and variables)**:** Consider an internal computation step

$$(h, \mathsf{v}, (\mu, mc) \circ \mathsf{CS}^b) \rightsquigarrow (h', \mathsf{v}', \acute{\mathsf{CS}}^b).$$

Let v_s be the restriction of v on exactly the variables which occur in the expressions e that have been evaluated or updated due to the above mentioned computation step. Further, let $h_s = h \downarrow_{\{\mu(\texttt{this}), [\![e_c]\!]_h^{\mathsf{v},\mu}\}}$ if the computation step is a method call and e_c is the callee expression, or $h_s = h \downarrow_{\{\mu(\texttt{this})\}}$ otherwise. Then also

$$(h_s, \mathsf{v}_s, (\mu, mc) \circ \mathsf{CS}^b) \rightsquigarrow (h_s', \mathsf{v}_s', \acute{\mathsf{CS}}^b),$$

such that $h_s' = h' \downarrow_{dom(h_s')}$ and $v_s' = v' \downarrow_{dom(v_s)}$.

Proof. Straightforward. The selection process regarding the necessary objects in the heap ensures that for all possible internal transitions all objects names which might be dereferenced, leading to a lookup in the heap, are included in the minimized heap. This ensures that the minimized configuration is enabled and since the internal computations are deterministic (modulo new object names), the statement then also follows from Lemma B.0.2. Note that the final heaps h' and h'_s are equal on the complete domain of h'_s which might include a new object name due to a constructor call. □

Lemma B.0.11 (Decomposition, single step)**:** Let $c, c' \in \mathit{Conf}$ such that $c \rightsquigarrow_p c'$ for some component p. Moreover, assume name contexts Δ, Θ and components p_1 and p_2 with $p_1 \Joinrel\Leftarrow p_2 = p$, $\Delta \vdash p_1 : \Theta$, and $\Theta \vdash p_2 : \Delta$ as well as configurations c_1 and c_2 with $c_1 \Joinrel\Leftarrow c_2 = c$, $\Delta \vdash c_1 : \Theta$, and $\Theta \vdash c_2 : \Delta$. Then one of the following properties hold:

1. There exists a communication label a such that $\Delta \vdash c_1 : \Theta \xrightarrow{a}_{p_1} \Delta' \vdash c'_1 : \Theta'$ and $\Theta \vdash c_2 : \Delta \xrightarrow{\bar{a}}_{p_2} \Theta' \vdash c'_2 : \Delta'$ with $c'_1 \Joinrel\Leftarrow c'_2 = c'$ or
2. $c_1 \rightsquigarrow_{p_1} c'_1$ such that $c'_1 \Joinrel\Leftarrow c_2 = c'$ or $c_2 \rightsquigarrow_{p_2} c'_2$ such that $c_1 \Joinrel\Leftarrow c'_2 = c'$.

Proof. By case analysis of the transition from c to c'. We show the most interesting cases.

Case simple transition

That is, let $c = (h, \mathsf{v}, \mathsf{AR}^a \circ \mathsf{CS}^b) \rightsquigarrow (h', \mathsf{v}', \acute{\mathsf{AR}}^a \circ \mathsf{CS}^b)$. Then AR^a is either part of the call stack of c_1 or of c_2. Let us assume without the loss of generality that $c_1 = (h_1, \mathsf{v}_1, \mathsf{AR}^a \circ \mathsf{CS}^b_1)$. It is $\mathsf{v}_1 \subset \mathsf{v}$ and since $\Delta \vdash c_1 : \Theta$ we also know that the topmost statement of AR^a does not involve the evaluation of variables of $\mathit{dom}(\mathsf{v}) \setminus \mathit{dom}(\mathsf{v}_1)$. This fact, together with Lemma B.0.1 and Lemma B.0.10 yields $c_1 \rightsquigarrow (h'_1, \mathsf{v}'_1, \acute{\mathsf{AR}}^a \circ \mathsf{CS}^b_1)$ such that $\mathit{dom}(h'_1) = \mathit{dom}(h') \downarrow_{\mathit{dom}(h_1)}$ and $\mathit{dom}(\mathsf{v}'_1) = \mathit{dom}(\mathsf{v}') \downarrow_{\mathit{dom}(h_1)}$. This leads to $(h'_1, \mathsf{v}'_1, \acute{\mathsf{AR}}^a \circ \mathsf{CS}^b_1) \Joinrel\Leftarrow c_2 = c'$.

Case internal method call: $\mathsf{AR}^a = (\mu, e.m(\overline{e}); mc)$

That is,

$$c = (h, \mathsf{v}, (\mu, e.m(\overline{e}); mc) \circ \mathsf{CS}^b) \rightsquigarrow_p (h, \mathsf{v}, \mathsf{AR}^a_m \circ (\mu, \texttt{rcv}\ x;\ mc) \circ \mathsf{CS}^b) = c',$$

where AR^a_m consists of the method body code of the method m. Let us assume that the calling activation record is part of c_1, i.e., $c_1 = (h_1, \mathsf{v}_1, (\mu, e.m(\overline{e}); mc) \circ \mathsf{CS}^b_1)$. Since c_1 is a well-typed configuration, it is $[\![e]\!]^{\mathsf{v}_1,\mu}_{h_1} = [\![e]\!]^{\mathsf{v},\mu}_{h}$ and we assume that the expression is evaluated to some object name o.

Subcase $o \in \mathit{dom}(h_1)$

The precondition of the lemma regarding c_1 and p_1 as well as Lemma B.0.10 and Lemma B.0.1 yield that also

$$\begin{aligned} c_1 &= (h_1, \mathsf{v}_1, (\mu, e.m(\overline{e}); mc) \circ \mathsf{CS}^b_1) \rightsquigarrow_{p_1} (h_1, \mathsf{v}_1, \mathsf{AR}^a_m \circ (\mu, \texttt{rcv}\ x;\ mc) \circ \mathsf{CS}^b_1) \\ &= c'_1, \end{aligned}$$

Assume $c_2 = (h_2, \mathsf{v}_2, \mathsf{CS}_2^b)$. Then from $c_1 \Joinrel\Supset c_2 = c$ and Lemma B.0.1 it follows that $\mathsf{CS}^b = f(\mathsf{CS}_1^b, \mathsf{CS}_2^b)$. And we get

$$\begin{aligned} c_1' \Joinrel\Supset c_2 &= (h_1, \mathsf{v}_1, \mathsf{AR}_m^a \circ (\mu, \texttt{rcv}\ x;\ mc) \circ \mathsf{CS}_1^b) \Joinrel\Supset (h_1, \mathsf{v}_2, \mathsf{CS}_2^b) \\ &= (h_1 \frown h_2, \mathsf{v}_1 \frown \mathsf{v}_2, \mathsf{AR}_m^a \circ (\mu, \texttt{rcv}\ x;\ mc) \circ f(\mathsf{CS}_1^b, \mathsf{CS}_2^b) \\ &= c_1 \end{aligned}$$

$\boxed{\textbf{Subcase}}$ $o \in dom(h_2)$

$$\begin{aligned} &\Delta \vdash c_1 : \Theta = \Delta \vdash (h_1, \mathsf{v}_1, (\mu, e.m(\overline{e}); mc) \circ \mathsf{CS}_1^b) : \Theta \xrightarrow{a}_{p_1} \\ &\Delta \vdash (h_1, \mathsf{v}_1, (\mu, \texttt{rcv}\ x{:}T;\ mc) \circ \mathsf{CS}_1^b) : \Theta, \Theta' = \Delta \vdash c_1' : \Theta, \Theta', \end{aligned}$$

where $a = \nu(\Theta').\langle call\ o.m(\overline{v})\rangle!$. On the other hand, the stack of c_2 is externally blocked. Moreover, p and p_2 share the same class definition of the class of o such that

$$\begin{aligned} &\Theta \vdash c_2 : \Delta = \Theta \vdash (h_2, \mathsf{v}_2, \mathsf{CS}_2^{eb}) : \Delta \xrightarrow{\bar{a}}_{p_2} \\ &\Theta, \Theta' \vdash (h_2, \mathsf{v}_2, \mathsf{AR}_m^a \circ \mathsf{CS}_2^{eb}) : \Delta = \Theta, \Theta' \vdash c_2' : \Delta. \end{aligned}$$

According to the definition of the stack merge it is

$$((\mu, \texttt{rcv}\ x{:}T;\ mc) \circ \mathsf{CS}_1^b) \mathbin{/\!\!\wedge\!\!\backslash} (\mathsf{AR}_m^a \circ \mathsf{CS}_2^{eb}) = \mathsf{AR}_m^a \circ (\mu, \texttt{rcv}\ x;\ mc) \circ (\mathsf{CS}_1^b \mathbin{/\!\!\wedge\!\!\backslash} \mathsf{CS}_2^{eb})$$

which proves the statement.

$\boxed{\boxed{\textbf{Case}}}$ internal return: $\mathsf{AR}^a = (\mu, \texttt{return}\ e;)$
That is,

$$c = (h, \mathsf{v}, (\mu, \texttt{return}\ e) \circ (\mu', \texttt{rcv}\ x;\ mc) \circ \mathsf{CS}^b) \rightsquigarrow_p (h, \mathsf{v}', (\mu'', mc) \circ \mathsf{CS}^b) = c',$$

Let us again assume that AR^a is part of the call stack of c_1. As for the second activation record there exist two possibilities; either it is also part of c_1 or in the call stack of c_2.

$\boxed{\textbf{Subcase}}$ receiving activation record is in c_2
Since c_1 has an active activation record on top and since $c_1 \Joinrel\Supset c_2$ is defined, the topmost activation record of c_2 must be externally blocked. Moreover, the merge of to call stacks does not change the order of the activation records. Thus, the second activation record of c is the topmost activation record of c_2 but annotated with the return type. As a consequence for both components we get the following transitions:

$$\begin{aligned} &\Delta \vdash c_1 : \Theta = \Delta \vdash (h_1, \mathsf{v}_1, (\mu, \texttt{return}\ e) \circ \mathsf{CS}_1^{eb}) : \Theta \xrightarrow{a}_{p_1} \\ &\Delta \vdash (h_1, \mathsf{v}_1, \mathsf{CS}_1^{eb}) : \Theta, \Theta' = \Delta \vdash c_1' : \Theta, \Theta' \end{aligned}$$

as well as

$$\begin{aligned} &\Theta \vdash c_2 : \Delta = \Theta \vdash (h_2, \mathsf{v}_2, (\mu', \texttt{rcv}\ x{:}T;\ mc) \circ \mathsf{CS}_2^b) : \Delta \xrightarrow{\bar{a}}_{p_2} \\ &\Theta, \Theta' \vdash (h_2, \mathsf{v}_2', (\mu'', mc) \circ \mathsf{CS}_2^b) : \Delta = \Theta, \Theta' \vdash c_2' : \Delta, \end{aligned}$$

where $a = \nu(\Theta').\langle return(v)\rangle!$. From $c_1 \Supset c_2 = c$ we can deduce that $\mathsf{CS}^b = f(\mathsf{CS}_2^b, \mathsf{CS}_1^{eb})$. Thus,

$$\mathsf{CS}_1^{eb} \mathbin{/\!\!\backslash\!\!\backslash} (\mu'', mc) \circ \mathsf{CS}_2^b = (\mu'', mc) \circ f(\mathsf{CS}_2^{eb}, \mathsf{CS}_2^b) = (\mu'', mc) \circ \mathsf{CS}^b,$$

which leads to $c_1' \Supset c_2' = c'$. Other cases are similar or dual. □

Finally, we can prove Compositionality-Lemma 2.5.5:

Proof. The proof follows directly by induction on the length of the transition sequence by applying Lemma B.0.8 and Lemma B.0.9, respectively, for the composition direction of the proof and Lemma B.0.11 for the decomposition direction. □

Appendix C

Code generation

C.1 Preprocessing

In this section, we want to show that preprocessing a specification results in a new specification such that the two specifications are behavioral equivalent regarding the interface communication. For this, as described in Section 4.4, we will provide a binary relation for which we will show that it represents a weak bisimulation. Furthermore, we will show that the pair of initial configurations of both specifications is included in the bisimulation relation. Recall, the preprocessing is basically done by means of two functions, $prep_{in}$ and $prep_{out}$ (cf. Table 4.2 and 4.1 in Section 4.1), which implement the preprocessing of passive and active statements, respectively. Hence the preprocessing functions are defined for *static* code, only. In order to define the bisimulation relation, we need to lift the preprocessing definition to *dynamic* code, namely to the code of activation records mc (cf. Section 3.4).

Definition C.1.1 (Preprocessed activation record code)**:** We extend range and domain of the preprocessing functions $prep_{in}$ and $prep_{out}$, originally defined in Section 4.1, to

$$prep_{out} : mc \to mc \quad \text{and} \quad prep_{in} : mc \times s_{nxt} \to s_{nxt} \times mc.$$

We additionally define

$$\begin{aligned} prep_{out}(s^{act};\ !ret;\ mc_1^{psv}) &\stackrel{\text{def}}{=} prep_{out}(s^{act});\ !ret;\ prep_{in}(mc_2^{psv}) \\ &\quad \text{with } (_, mc_2^{psv}) = prep_{in}(mc_1^{psv}, success) \end{aligned}$$

as well as

$$\begin{aligned} prep_{in}(s_1^{psv};\ x{=}?ret;\ mc^{act}, s_{nxt}) &\stackrel{\text{def}}{=} (s'_{nxt},\ s_2^{psv};\ [i]x{=}?ret;\ check(i, e');\\ &\qquad prep_{out}(mc^{act})) \\ &\quad \text{with } (s'_{nxt}, s_2^{psv}) = prep_{in}(s_1^{psv}, next = i), \end{aligned}$$

where $!ret$ and $?ret$ abbreviate $!\texttt{return}(e)$ and $?\texttt{return}(T\ x').\texttt{where}(e)$, respectively.

Based on the definition above, we can define the bisimulation relation R_b. The idea is to relate each configuration of the original specification with the corresponding specification of the preprocessed specification. Thus, as for the heap and the global variables, we relate configurations which are almost identical but where the configurations of the preprocessed specification only provides the additional global variable *next* which stores an arbitrary expectation identifier i. Regarding the activation record code of configuration pairs of R_b, we basically relate code to its preprocessed variant according to the preprocessing functions of Definition C.1.1. An exception is code mc^{act} whose preprocessed variant starts with an *next* update statement s_{nxt}. For instance, the preprocessing of an outgoing call statement results in a corresponing call statement but which is preceded by an update statement. In these case we have to relate the original mc^{act} code not only to the preprocessing result but additionally to all the code that result from reducing s_{nxt} in terms of internal steps.

Definition C.1.2 (Bisimulation relation R_b)**:** We define a binary relation $R_b \subset Conf \times Conf$ over configurations of the specification language, such that for all heap functions h, global variable functions v, local variable function lists μ, and activation record code mc_1^{act} or, respectively, mc_1^{psv} exactly the following pairs are included:

1. $$((h, \mathsf{v}, (\mu, mc_1^{psv})), (h, \mathsf{v}_{+[next]}, (\mu, mc_2^{psv}))) \in R_b,$$

   if $(_, mc_2^{psv}) = prep_{in}(mc_1^{psv}, success)$.

2. $$((h, \mathsf{v}, (\mu, mc_1^{act})), (h, \mathsf{v}_{+[next]}, (\mu, mc_2^{act}))) \in R_b,$$

   $$\text{if } mc_2^{act} = \begin{cases} s'_{nxt};\ mc^{act} & \text{if } prep_{out}(mc_1^{act}) = s_{nxt};\ mc^{act} \text{ with} \\ & \quad (h, \mathsf{v}_{+[next]}, (\mu, s_{nxt})) \rightsquigarrow^* (h, \mathsf{v}_{+[next]}, (\mu, s'_{nxt})). \\ prep_{out}(mc_1^{act}) & \text{else} \end{cases}$$

where $\mathsf{v}_{+[next]}$ represents the variable function that extends v with *next* such that *next* stores an arbitrary expectation identifier. In particular, v must not include a variable with this name, already. And correspondingly, mc_1^{act} and mc_1^{psv} must not include references to a variable *next*.

Note, according to the definition, R_b does not define a function. Instead, for each configuration c_1 with $(c_1, c_2) \in R_b$ for some configuration c_2, there exist several other configurations $c_3 \neq c_2$ such that also $(c_1, c_3) \in R_b$. For, on the one hand, the right hand side configuration may vary in the value of the global variable *next*. On the other hand, as mentioned above already, if c_1's activation record code is preprocessed resulting into code that starts with an update statement s_{nxt}, then c_1 is not only related to configurations that provide the corresponding preprocessed code but also to its successors where s_{nxt} has been reduced already.

Finally, we have to prove that the relation R_b is indeed a weak bisimulation relation. This is stated in the following lemma.

Lemma C.1.3: The binary relation R_b given in Definition C.1.2 represents a weak bisimulation in the sense of Definition 4.4.4.

Proof. Assume two configurations $c_1, c_2 \in \mathit{Conf}$ with $(c_1, c_2) \in R_b$. The definition of R_b implies that there exist a heap function h, a global variable function v, a local variable function list μ, and activation record code mc such that c_1 is of the form

$$c_1 = (h, \mathsf{v}, (\mu, mc))$$

and c_2 is of the form

$$c_2 = (h, \mathsf{v}_{+[next]}, (\mu, mc')),$$

where mc' corresponds to mc_2^{psv} or mc_2^{act} of Definition C.1.2. We prove the lemma by means of a case analysis regarding the construction of mc of the configuration c_1. In particular, for each case we will show both simulation directions at the same time. That is, in each case, we will prove that

- on the one hand, for each possible transition steps of c_1 to c_1'

$$\begin{array}{ccc} c_1 \rightsquigarrow c_1' & \text{implies} & c_2 \rightsquigarrow^* c_2' \\ & \text{and} & \\ \Delta \vdash c_1 : \Theta \xrightarrow{a} \Delta' \vdash c_1' : \Theta' & \text{implies} & \Delta \vdash c_2 : \Theta \stackrel{a}{\Longrightarrow} \Delta' \vdash c_2' : \Theta' \end{array}$$

- and, on the other hand, for each possible transition steps of c_2 to c_2'

$$\begin{array}{ccc} c_2 \rightsquigarrow c_2' & \text{implies} & c_1 \rightsquigarrow^* c_1' \\ & \text{and} & \\ \Delta \vdash c_2 : \Theta \xrightarrow{a} \Delta' \vdash c_2' : \Theta' & \text{implies} & \Delta \vdash c_1 : \Theta \stackrel{a}{\Longrightarrow} \Delta' \vdash c_1' : \Theta', \end{array}$$

such that in all cases $(c_1', c_2') \in R_b$. Within the proof we will refer to the firstly mentioned direction (i.e., c_2 simulates c_1) by using the right arrow $\Rightarrow$ and correspondingly to the lastly mentioned direction (i.e., c_1 simulates c_2) by using the left arrow $\Leftarrow$. We show some exemplary cases only as the remaining cases are similar. Note that according to the operational semantics each starting configuration only allows for either an internal or an external transition step.

Case $mc = \texttt{if}(e)\ \{s_1^{act}\}\ \texttt{else}\ \{s_2^{act}\};\ s^{act}$
In this case we have

$$mc' = \texttt{if}(e)\ \{\mathit{prep}_{out}(s_1^{act})\}\ \texttt{else}\ \{\mathit{prep}_{out}(s_2^{act})\};\ \mathit{prep}_{out}(s^{act})$$

according to Definition C.1.1 and to the sequential and the conditional case of Table 4.1.

Direction $\Rightarrow$
We have to show that $c_1 \rightsquigarrow c_1'$ implies $c_2 \rightsquigarrow^* c_2'$, as c_1 can only be reduced by an internal transition. Specifically, the rules $\textsc{Cond}_1$ and, respectively, $\textsc{Cond}_2$

regarding the internal steps of the specification language's operational semantics yield

$$c_1 \rightsquigarrow c_1' \quad \text{with}$$
$$c_1' = (h, \mathsf{v}, (\mu, s_1^{act};\ s^{act})) \quad \text{or} \quad c_1' = (h, \mathsf{v}, (\mu, s_2^{act};\ s^{act})),$$

respectively, depending on the evaluation of $[\![e]\!]_h^{\mu,\mathsf{v}}$. Correspondingly, we get

$$\begin{aligned} &c_2 \rightsquigarrow c_2' \quad \text{with} \\ &c_2' = (h, \mathsf{v}_{+[next]}, (\mu, prep_{out}(s_1^{act});\ prep_{out}(s^{act}))) \quad \text{or} \\ &c_2' = (h, \mathsf{v}_{+[next]}, (\mu, prep_{out}(s_2^{act});\ prep_{out}(s^{act}))). \end{aligned}$$

According to the definition of $prep_{out}$ for the sequential composition, it is

$$(c_1', c_2') \in R_b.$$

Direction $\Leftarrow$
Also c_2 can only be reduced by means of an internal transition, so we have to show that $c_2 \rightsquigarrow c_2'$ implies $c_1 \rightsquigarrow^* c_1'$. Again, we can only apply rule $\textsc{Cond}_1$ or $\textsc{Cond}_2$, if $[\![e]\!]_h^{\mu,\mathsf{v}_{+[next]}}$ evaluates to *true* or to *false*, respectively. Since e must not contain any references to *next*, it is

$$[\![e]\!]_h^{\mu,\mathsf{v}_{+[next]}} = [\![e]\!]_h^{\mu,\mathsf{v}}.$$

Hence, $c_1 \rightsquigarrow c_1'$ where c_1' and c_2' are of the same form as in the above proof regarding the other direction. Therefore, again, it is $(c_1', c_2') \in R_b$.

Case $mc = x = e; s^{act}$
As for c_2, it is $mc' = x = e;\ prep_{out}(s^{act})$. Thus, the first statement of c_1's code and of c_2's code is the same assignment and so it is easy to see that

$$c_1 \rightsquigarrow c_1' \text{ implies that } c_2 \rightsquigarrow c_2',$$

but also conversely,

$$c_2 \rightsquigarrow c_2' \text{ implies that } c_1 \rightsquigarrow c_1',$$

such that, regarding both proof directions

$$(c_1', c_2') \in R_b.$$

Case $mc = e!m(\overline{e})\ \{\ \overline{T}\,\overline{x};\ s_1^{psv};\ x = ?\texttt{return}(T\,x').\texttt{where}(e')\ \};\ s^{act}$
Then regarding the activation record code of c_2, the definition of R_b allows for the following possibilities. Either it is

$$\begin{aligned} mc' = {}& s_{nxt}';\ e!m(\overline{e})\ \{\ \overline{T}\,\overline{x};\ s_2^{psv};\ [i]\,x = ?\texttt{return}(T\,x').\texttt{where}(e')\ \}; \\ & check(i, e');\ prep_{out}(s^{act}), \end{aligned}$$

or, similarly, but without the preceding update statement, it is

$$\begin{aligned} mc' = \; & e!m(\overline{e})\ \{\ \overline{T}\,\overline{x};\ s_2^{psv};\ [i]\,x = ?\texttt{return}(T\,x').\texttt{where}(e')\ \};\\ & check(i,e');\ prep_{out}(s^{act}), \end{aligned}$$

with $(*)$ $(s_{nxt}, s_2^{psv}) = prep_{in}(s_1^{psv}, next = i)$ and

$$(h, \mathsf{v}_{+[next]}, (\mu, s_{nxt})) \rightsquigarrow^* (h, \mathsf{v}_{+[next]}, (\mu, s'_{nxt})).$$

$\boxed{\textbf{Direction}}$ $\Rightarrow$
The configuration c_1 can only be reduced by an outgoing method call. Therefore, for appropriate name contexts Δ, Δ', Θ and an outgoing method call label a it is

$$\Delta \vdash c_1 : \Theta \xrightarrow{a} \Delta' \vdash c'_1 : \Theta,$$

where the configuration c'_1 is of the form

$$c'_1 = (h, \mathsf{v}, (\mu', s_1^{psv};\ x = ?\texttt{return}(T\,x').\texttt{where}(e')\ \};\ s^{act}))$$

according to the rule CallO of the external semantics. As for c_2, if need be, we first process the update statement s'_{nxt} by internal transitions, so we get

$$\begin{aligned} c_2 \rightsquigarrow^* c'_2 = (\ h, \mathsf{v}_{+[next]},\ & e!m(\overline{e})\ \{\ \overline{T}\,\overline{x};\ s_2^{psv};\ [i]\,x = ?\texttt{return}(T\,x').\texttt{where}(e')\ \};\\ & check(i,e');\ prep_{out}(s^{act})\ ), \end{aligned}$$

where the global variable function of c'_2 has only changed the value of *next*. Furthermore, the external semantics yields

$$\Delta \vdash c'_2 : \Theta \xrightarrow{a} \Delta' \vdash c''_2 : \Theta,$$

such that

$$c''_2 = (h, \mathsf{v}'_{+[next]}, s_2^{psv};\ [i]\,x = ?\texttt{return}(T\,x').\texttt{where}(e');\ check(i,e');\ prep_{out}(s^{act})).$$

Due to the equation $(*)$ and according to Definition C.1.1 it is

$$(c'_1, c''_2) \in R_b.$$

$\boxed{\textbf{Direction}}$ $\Leftarrow$
If mc' starts with an update statement s'_{nxt} then

$$c_2 \rightsquigarrow c'_2$$

such that $(c_1, c'_2) \in R_b$. Alternatively, as shown above, the first statement of mc' can be an outgoing call statement. In this case, c_2 equals the configuration c'_2 of the other proof direction that we have discussed above already. Due to the fact, that expressions in mc' must not include references to the extra variable *next*, all outgoing call labels a, involved in a transition from c'_2 to c''_2, can also be applied to c_1 such that, again, $\Delta \vdash c_1 : \Theta \xrightarrow{a} \Delta' \vdash c'_1 : \Theta$ such that$(c'_1, c''_2) \in R_b$.

Case $mc = \texttt{if}(e)\ \{s_1^{psv}\}\ \texttt{else}\ \{s_2^{psv}\};\ s^{psv}$
According to the definition of $prep_{in}$ in Table 4.2, it is

$$mc' = \texttt{if}(e)\ \{\tilde{s}_1^p\}\ \texttt{else}\ \{\tilde{s}_2^p\};\ \tilde{s}^p,$$

where $(s_{nxt}, \tilde{s}^p) = prep_{in}(s^{psv}, success)$ and, for each $i \in \{1, 2\}$,

$$(_, \tilde{s}_i^p) = prep_{in}(s_i^{psv}, s_{nxt}).$$

Direction $\Rightarrow$
According to the operational semantics, only the internal rules $\textsc{Cond}_1$ or $\textsc{Cond}_2$ can be applied, in order to reduce the configuration c_1: if $[\![e]\!]_h^{\mu,\mathsf{v}}$ evaluates to *true* or to *false*, then $c_1 \rightsquigarrow c_1'$ such that

$$c_1' = (h, \mathsf{v}, (\mu, s_1^{psv};\ s^{psv})) \quad \text{or, resp.,} \quad c_1' = (h, \mathsf{v}, (\mu, s_2^{psv};\ s^{psv})).$$

Correspondingly, we get $c_2 \rightsquigarrow c_2'$ with

$$c_2' = (h, \mathsf{v}_{+[next]}, (\mu, \tilde{s}_1^p;\ \tilde{s}^p)) \quad \text{or, resp.,} \quad c_2' = (h, \mathsf{v}_{+[next]}, (\mu, \tilde{s}_2^p;\ \tilde{s}^p)).$$

The definition of $prep_{in}$ regarding sequential compositions yields in both cases

$$(c_1', c_2') \in R_b.$$

Direction $\Leftarrow$
Both configurations, c_1 and c_2, can only be reduced by one of the internal rules $\textsc{Cond}_1$ or $\textsc{Cond}_2$. Moreover, recall again that e must not depend on the value of *next*. Therefore, the proof that we have given for the other direction also represents a proof for this direction.

Case $mc = (C\ x)?(\overline{T}\ \overline{x}).\texttt{where}(e)\{\overline{T_l}\ \overline{x_l};\ s^{act};\ !\texttt{return}(e')\};\ s^{psv}$
Again, according to the definition of $prep_{in}$, the activation record code of c_2 is

$$mc' = [i]\,(Cx)?(\overline{T}\overline{x}).\texttt{where}(e)\{\overline{T_l}\overline{x_l};\ check(i, e);\ prep_{out}(s^{act});\ s_{nxt};\ !\texttt{return}(e')\};\ \tilde{s}^p,$$

with $(s_{nxt}, \tilde{s}^p) = prep_{in}(s^{psv}, success)$.

Direction $\Rightarrow$
The configuration c_1 allows for external transition steps only. In particular, it only allows transitions which are labeled with an incoming call label a such that

$$\Delta \vdash c_1 : \Theta \xrightarrow{a} \Delta' \vdash c_1' : \Theta,$$

with

$$c_1' = (h, \mathsf{v}, (\mu', s^{act};\ !\texttt{return}(e');\ s^{psv}))$$

The configuration c_2 allows for the same transition step. Specifically, it is

$$\Delta \vdash c_2 : \Theta \xrightarrow{a} \Delta' \vdash c_2' : \Theta,$$

where

$$c_2' = (h, \mathsf{v}_{+[next]}, (\mu', check(i,e);\ prep_{out}(s^{act});\ s_{nxt}; !\texttt{return}(e')\};\ \tilde{s}^p))$$

and, since we assume $check(i,e)$ to equal ϵ, additionally

$$c_2' \rightsquigarrow^* c_2'' = (h, \mathsf{v}_{+[next]}, (\mu', prep_{out}(s^{act});\ s_{nxt}; !\texttt{return}(e')\};\ \tilde{s}^p)).$$

According to the definition of $prep_{in}$ and the definition of $\tilde{s}^p$, we get

$$(c_1, c_2'') \in R_b.$$

Direction $\Leftarrow$

Also for c_2 the operational semantics permits only incoming method call steps a such that

$$\Delta \vdash c_2 : \Theta \xrightarrow{a} \Delta' \vdash c_2' : \Theta,$$

where

$$c_2' = (h, \mathsf{v}_{+[next]}, (\mu', check(i,e);\ prep_{out}(s^{act});\ s_{nxt}; !\texttt{return}(e')\};\ \tilde{s}^p)).$$

Again, equating $check(i,e)$ with ϵ we can further say

$$c_2' = (h, \mathsf{v}_{+[next]}, (\mu', prep_{out}(s^{act});\ s_{nxt}; !\texttt{return}(e')\};\ \tilde{s}^p)).$$

Finally, regarding the same name contexts and the same communication label, we get

$$\Delta \vdash c_1 : \Theta \xrightarrow{a} \Delta' \vdash c_1' : \Theta,$$

with

$$c_1' = (h, \mathsf{v}, (\mu', s^{act};\ !\texttt{return}(e');\ s^{psv})).$$

And again according to the definition of $prep_{in}$ and the definition of $\tilde{s}^p$, we can conclude

$$(c_1, c_2'') \in R_b.$$

□

Lemma C.1.4: Assume a specification s with $\Delta \vdash s : \Theta$. Additionally, consider a specification s' that results from s by adding the global *next* variable and by preprocessing its main statement. Then

$$(c_{init}(s), c_{init}(s')) \in R_b.$$

Proof. Consider

$$s = \overline{cutdecl}\ \overline{T}\ \overline{x};\ \overline{mokdecl}\ \{stmt;\ \texttt{return}\},$$

to be a valid specification. Further, assume a specification s' such that

$$s' = \overline{cutdecl}\ \overline{T}\ \overline{x};\ T\ next;\ \overline{mokdecl}\ \{stmt';\ \texttt{return}\},$$

where $stmt'$ results from either applying $prep_{in}$ or $prep_{out}$ to $stmt$, depending on the control context of the statement. Then the claim immediately follows from the Definition C.1.2 of R_b. □

C.2 Anticipation

In order to prove that the first preprocessing step indeed represents an anticipation mechanism of the expected interface communication, we first introduce some auxiliary definitions.

Definition C.2.1 (Anticipation-valid code)**:** The code mc of an activation record is said to be anticipation-valid if there exist update-statements $s_{nxt}^{`}$ and $s_{nxt}^{'}$ such that the judgment $s_{nxt}^{`} \vdash_{\mathsf{as}} mc : s_{nxt}^{'}$ is deducible according to the inference rules given in Table C.1.

Lemma C.2.2: Static anticipation-validity implies proper anticipation:

1. Assume $s_{nxt}^{`} \vdash_{\mathsf{as}} mc^{psv} : s_{nxt}^{'}$. Then for all heaps h, all global variable functions v, and all local variable function lists μ the following holds. If

$$(h, \mathsf{v}, (\mu, s_{nxt}^{`})) \rightsquigarrow^* (h, \mathsf{v}, (\mu, next = i))$$

and

$$(h, \mathsf{v}, (\mu, mc^{psv})) \rightsquigarrow^* (h, \mathsf{v}, (\mu, [j]\, mc^{psv\prime}).$$

then $i = j$.

2. Assume $s_{nxt}^{`} \vdash_{\mathsf{as}} mc^{act} : s_{nxt}^{'}$. Then for all heaps h, all global variable functions v, and all local variable function lists μ the following holds. If

$$(h, \mathsf{v}, (\mu, s_{nxt}^{`})) \rightsquigarrow^* (h, \mathsf{v}, (\mu, next = i))$$

and

$$(h, \mathsf{v}, (\mu, mc^{act})) \xrightarrow{\gamma!} (h, \mathsf{v}, (\mu, [j]\, mc^{psv\prime}).$$

then $i = j$.

Proof. Both, the passive and the active case will be proven by induction on the construction of the code. Let us first assume that

$$s_{nxt}^{`} \vdash_{\mathsf{as}} mc^{psv} : s_{nxt}^{'} \quad \text{and} \quad (h, \mathsf{v}, (\mu, s_{nxt}^{`})) \rightsquigarrow^* (h, \mathsf{v}, (\mu, next = i))$$

for some heap h, global variable function v, and local variable function list μ. We do a case analysis regarding the code:

Case $mc^{psv} = [i]\,(C\ x)?m(\overline{T}\,\overline{x}).\texttt{where}(e')\{\overline{T_l}\,\overline{x_l};\ s^{act};\ s_{nxt};\ !\texttt{return}(e)\}$
According to Rule AS-CallIn it is $s_{nxt}^{`} = next == i$. Thus trivially the proposition holds.

Case $mc^{psv} = \epsilon$
Nothing to show, as ϵ does not evolve to an incoming call or incoming return statement.

Case $mc^{psv} = s_1^{psv}; s_2^{psv}$
The proof for this case follows from the induction hypothesis and the premises $s_{nxt}^{`} \vdash_{\mathsf{as}} s_1^{psv} : s_{nxt}$ and $s_{nxt} \vdash_{\mathsf{as}} s_2^{psv} : s_{nxt}^{'}$ of Rule AS-Seq^p. However, we have to distinguish two sub-cases.

$$[\text{AS-CallIn}]\ \frac{_ \vdash_{\mathsf{as}} s^{act} : _ \quad s_{nxt} = s'_{nxt}}{next{=}i \vdash_{\mathsf{as}} [i]\,(C\ x)?m(\overline{T}\ \overline{x}).\texttt{where}(e')\{\overline{T_l}\ \overline{x_l};\ s^{act};\ s_{nxt};\ !\texttt{return}(e)\} : s'_{nxt}}$$

$$[\text{AS-Seq}^p]\ \frac{s'_{nxt} \vdash_{\mathsf{as}} s_1^{psv} : s_{nxt} \quad s_{nxt} \vdash_{\mathsf{as}} s_2^{psv} : s'_{nxt}}{s'_{nxt} \vdash_{\mathsf{as}} s_1^{psv}; s_2^{psv} : s'_{nxt}}$$

$$[\text{AS-While}^p]\ \frac{s_{nxt} \vdash_{\mathsf{as}} s^{psv} : s'_{nxt} \quad s'_{nxt} = \texttt{if}(e)\ \{s_{nxt}\}\ \texttt{else}\ \{s'_{nxt}\}}{s'_{nxt} \vdash_{\mathsf{as}} \texttt{while}(e)\ \{s^{psv}\} : s'_{nxt}}$$

$$[\text{AS-If}^p]\ \frac{s_{nxt1} \vdash_{\mathsf{as}} s_1^{psv} : s'_{nxt} \quad s_{nxt2} \vdash_{\mathsf{as}} s_2^{psv} : s'_{nxt} \quad s'_{nxt} = \texttt{if}(e)\ \{s_{nxt1}\}\ \texttt{else}\ \{s_{nxt2}\}}{s'_{nxt} \vdash_{\mathsf{as}} \texttt{if}(e)\ \{s^{psv}{}_1\}\ \texttt{else}\ \{s^{psv}{}_2\} : s'_{nxt}}$$

$$[\text{AS-Case}]\ \frac{next = i \vdash_{\mathsf{as}} [i]\ \overline{stmt_{in}; s^{psv}} : s'_{nxt}}{next = i \vdash_{\mathsf{as}} \texttt{case}\ [i]\ \overline{stmt_{in}; s^{psv}} : s'_{nxt}}$$

$$[\text{AS-Skip}]\ \frac{s'_{nxt} = s'_{nxt}}{s'_{nxt} \vdash_{\mathsf{as}} \epsilon : s'_{nxt}}$$

$$[\text{AS-}s^{psv}\text{-RetIn}]\ \frac{s'_{nxt} \vdash_{\mathsf{as}} s^{psv} : next = i \quad _ \vdash_{\mathsf{as}} mc^{act} : s'_{nxt}}{s'_{nxt} \vdash_{\mathsf{as}} s^{psv};\ [i]\,x = ?\texttt{return}(T\ x').\texttt{where}(e);\ mc^{act} : s'_{nxt}}$$

$$[\text{AS-CallOut}]\ \frac{s_{nxt} \vdash_{\mathsf{as}} s^{psv} : next = i}{s'_{nxt} \vdash_{\mathsf{as}} s_{nxt};\ e!m(\overline{e})\{\overline{T_l}\ \overline{x_l}; s^{psv}; [i]\,x = ?\texttt{return}(T\ x').\texttt{where}(e)\} : next{=}i}$$

$$[\text{AS-Seq}^a]\ \frac{s'_{nxt} \vdash_{\mathsf{as}} s_1^{act} : s_{nxt} \quad s_{nxt} \vdash_{\mathsf{as}} s_2^{act} : s'_{nxt}}{s'_{nxt} \vdash_{\mathsf{as}} s_1^{act}; s_2^{act} : s'_{nxt}}$$

$$[\text{AS-While}^a]\ \frac{s'_{nxt} \vdash_{\mathsf{as}} s^{act} : s'_{nxt}}{s'_{nxt} \vdash_{\mathsf{as}} \texttt{while}(e)\ \{s^{act}\} : s'_{nxt}}$$

$$[\text{AS-If}^a]\ \frac{s'_{nxt} \vdash_{\mathsf{as}} s_1^{act} : s'_{nxt} \quad s'_{nxt} \vdash_{\mathsf{as}} s_2^{act} : s'_{nxt}}{s'_{nxt} \vdash_{\mathsf{as}} \texttt{if}(e)\ \{s_1^{act}\}\ \texttt{else}\ \{s_2^{act}\} : s'_{nxt}}$$

$$[\text{AS-}s^{act}\text{-RetOut}]\ \frac{s'_{nxt} \vdash_{\mathsf{as}} s^{act} : _ \quad s_{nxt} \vdash_{\mathsf{as}} mc^{psv} : s'_{nxt}}{s'_{nxt} \vdash_{\mathsf{as}} s^{act};\ s_{nxt};\ !\texttt{return}(e); mc^{psv} : s'_{nxt}}$$

$$[\text{AS-VUpd}]\ s'_{nxt} \vdash_{\mathsf{as}} x = e : s'_{nxt}$$

Table C.1: Anticipation-valid code (static)

$\boxed{\textbf{Subcase}}$ $s_1^{psv} = \epsilon$
Then $\grave{s_{nxt}} = s_{nxt}$ and $(h, \mathsf{v}, (\mu, s_1^{psv}; s_2^{psv})) \rightsquigarrow (h, \mathsf{v}, (\mu, s_2^{psv}))$, so the proposition follows from the hypothesis regarding s_2^{psv}.

$\boxed{\textbf{Subcase}}$ $s_1^{psv} \neq \epsilon$
In this case the proposition immediately follows from the induction hypothesis regarding s_1^{psv}.

$\boxed{\textbf{Case}}$ $mc^{psv} = \texttt{while}(e)\ \{s^{psv}\}$
According to Rule AS-WHILE^p it is $\grave{s_{nxt}} = \texttt{if}(e)\ \{s_{nxt}\}\ \texttt{else}\ \{\acute{s_{nxt}}\}$ with s_{nxt} such that $s_{nxt} \vdash_{\mathsf{as}} s^{psv} : \grave{s_{nxt}}$. Assume that $(h, \mathsf{v}, (\mu, s_{nxt})) \rightsquigarrow^* (h, \mathsf{v}, (\mu, next == i)$. The hypothesis yields $(h, \mathsf{v}, (\mu, s^{psv})) \rightsquigarrow^* (h, \mathsf{v}, (\mu, [i]\, mc^{psv\prime}))$. Assume h, v, and μ such that $[\![e]\!]_h^{\mathsf{v},\mu} = \mathit{true}$. Then

$$(h, \mathsf{v}, (\mu, \grave{s_{nxt}})) \rightsquigarrow (h, \mathsf{v}, (\mu, s_{nxt})) \rightsquigarrow^* (h, \mathsf{v}, (\mu, next == i)).$$

as well as

$$(h, \mathsf{v}, (\mu, \texttt{while}(e)\ \{s^{psv}\})) \rightsquigarrow (h, \mathsf{v}, (\mu, s^{psv}; \texttt{while}(e)\{s^{psv}\})) \rightsquigarrow^* (h, \mathsf{v}, (\mu, [i]\, mc^{psv\prime\prime})).$$

On the other hand, now consider the case that $[\![e]\!]_h^{\mathsf{v},\mu} = \mathit{false}$. Then we get

$$(h, \mathsf{v}, (\mu, \grave{s_{nxt}})) \rightsquigarrow (h, \mathsf{v}, (\mu, s_{nxt})) \rightsquigarrow^* (h, \mathsf{v}, (\mu, next == i)).$$

The remaining cases are similar.
Now let us assume that

$$\grave{s_{nxt}} \vdash_{\mathsf{as}} mc^{act} : \acute{s_{nxt}} \quad \text{and} \quad (h, \mathsf{v}, (\mu, \grave{s_{nxt}})) \rightsquigarrow^* (h, \mathsf{v}, (\mu, next = i))$$

for some heap h, global variable function v, and local variable function list μ. We do a case analysis regarding the code:

$\boxed{\textbf{Case}}$ $mc^{act} = s_{nxt};\ e!m(\overline{e})\{\overline{T_l}\ \overline{x_l}; s^{psv}; [i]\, x = ?\texttt{return}(T\ x').\texttt{where}(e)\}$
Due to the premise of Rule AS-CALLOUT the proposition follows from the passive case of this lemma.

$\boxed{\textbf{Case}}$ $mc^{act} = s_1^{act}; s_2^{act}$
Like in the passive case we have to distinguish two sub-cases: if s_1^{act} is the empty statement or a variable update then the proposition follows from the hypothesis of the second statement. Otherwise it follows from the hypothesis of the first statement.

Again the remaining cases are straightforward.

□

While the previous deduction system checks that some code anticipates the incoming communication expectations in the context of any configuration state, the next definition in contrast captures the anticipation feature within the context of a given state.

$$[\text{AD-}s^{psv}\text{-RetI}]\frac{h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} s^{psv} : next = i \quad _ \vdash_{\mathsf{as}} mc^{act} : s_{nxt}}{h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} s^{psv};\ [i]\, x = ?\mathtt{return}(T\ x').\mathtt{where}(e); mc^{act} : s_{nxt}}$$

$$[\text{AD-RetI}]\frac{[\![next]\!]_h^{\mathsf{v},\mu} = i \quad _ \vdash_{\mathsf{as}} mc^{act} : s_{nxt}}{h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} [i]\, x = ?\mathtt{return}(T\ x').\mathtt{where}(e); mc^{act} : s_{nxt}}$$

$$[\text{AD-}s^{psv}]\frac{_ \vdash_{\mathsf{as}} s^{psv} : s_{nxt} \quad [\![next]\!]_h^{\mathsf{v},\mu} = i \quad (h, \mathsf{v}, (\mu, s^{psv})) \rightsquigarrow^* [i]\, stmt_{in}}{h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} s^{psv} : s_{nxt}}$$

$$[\text{AD-}s^{act}\text{-RetOut}]\frac{h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} s^{act} : _\ s_{nxt} \vdash_{\mathsf{as}} mc^{psv} : s'_{nxt}}{h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} s^{act};\ s_{nxt};\ !\mathtt{return}(e);\ mc^{psv} : s'_{nxt}}$$

$$[\text{AD-RetOut}]\frac{h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc^{psv} : s_{nxt}}{h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} !\mathtt{return}(e);\ mc^{psv} : s_{nxt}}$$

$$[\text{AD-}stmt_{out}]\frac{h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} s^{psv} : next = i \quad _ \vdash_{\mathsf{as}} s^{act} : s_{nxt}}{h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} e!m(\overline{e})\{\overline{T_l}\ \overline{x_l}; s^{psv}; [i]\, x = ?\mathtt{return}(T\ x').\mathtt{where}(e)\}; s^{act} : s_{nxt}}$$

$$[\text{AD-}s_{nxt}]\frac{(h, \mathsf{v}, (\mu, s'_{nxt})) \rightsquigarrow^* (h, \mathsf{v}', (\mu, \epsilon)) \quad h, \mathsf{v}', \mu \vdash_{\mathsf{ad}} s^{act} : s_{nxt}}{h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} s'_{nxt};\ s^{act} : s_{nxt}}$$

$$[\text{AD-}s^{act}]\frac{s^{act} \neq stmt_{out}; s_2^{act} \quad _ \vdash_{\mathsf{as}} s^{act} : s_{nxt}}{h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} s^{act} : s_{nxt}}$$

Table C.2: Anticipation-valid configurations (dynamic)

Definition C.2.3 (Anticipation-valid configuration)**:** Assume a configuration

$$(h, \mathsf{v}, (\mu, mc)) \in Conf$$

of the specification language. Then we say that the configuration is *anticipation-valid*, written

$$h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc : \mathsf{anticip},$$

if the judgment $h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc : s_{nxt}$ can be derived for some update statement s_{nxt} by means of the deduction rules given in Table C.2 and Table C.1.

Lemma C.2.4 (Anticipation preprocessing establishes anticipation-validity)**:** Assume a statement $stmt$ such that $\Delta \vdash stmt\ : \Theta$. Furthermore, for a given update statement $s_{nxt}^{\acute{}}$ let $stmt' = prep(stmt, s_{nxt}^{\acute{}})$. Then for some appropriate update statement $s_{nxt}^{\grave{}}$ also $s_{nxt}^{\grave{}} \vdash_{\mathsf{as}} stmt' : s_{nxt}^{\acute{}}$ holds.

Proof. More specifically, we will prove in the following that, if $stmt$ is an instance of s^{psv} then it is $s_{nxt}^{\grave{}} \vdash_{\mathsf{as}} stmt' : s_{nxt}^{\acute{}}$ with $s_{nxt}^{\grave{}}$ defined by $(s_{nxt}^{\grave{}}, stmt') = prep_{in}(stmt, s_{nxt}^{\acute{}})$. Moreover, if $stmt$ is an instance of s^{act} we will show that

$_ \vdash_{\mathsf{as}} stmt' : _$ holds. By structural induction. We will show the interesting subcases for both cases, i.e., passive and active statement.

Case $stmt = s^{psv}$

In this case let us define $(s^{`}_{nxt}, s^{psv\prime}) = prep_{in}(s^{psv}, s^{\acute{}}_{nxt})$.

Subcase $stmt = (C\ x)?m(\overline{T}\ \overline{x}).\texttt{where}(e)\{\overline{T_l}\ \overline{x_l};\ s^{act};\ !\texttt{return}(e)\}$

According to the definition of $prep_{in}$ it is

$$\begin{aligned} s^{`}_{nxt} &= next = i \quad \text{and} \\ stmt' &= [i]\,(C\ x)?m(\overline{T}\ \overline{x}.\texttt{where}(e))\{\overline{T_l}\ \overline{x_l};\ s^{act\prime};\ s^{\acute{}}_{nxt};\ !\texttt{return}(e)\} \quad \text{with} \\ s^{act\prime} &= prep_{out}(s^{act}). \end{aligned}$$

The induction hypothesis implies $_ \vdash_{\mathsf{as}} s^{act\prime} : _$. Thus, both premises of Rule AS-CallIn are satisfied which proves the proposition.

Subcase $stmt = \mathsf{if}(e)\ \{s_1^{psv}\}\ \texttt{else}\ \{s_2^{psv}\}$

According to the definition of $prep_{in}$ it is

$$\begin{aligned} s^{`}_{nxt} &= \texttt{if}(e)\ \{s_{nxt1}\}\ \texttt{else}\ \{s_{nxt2}\} \text{ with} \\ (s_{nxt1}, s_1^{psv\prime}) &= prep_{in}(s_1^{psv}, s^{\acute{}}_{nxt}) \text{ and} \\ (s_{nxt2}, s_2^{psv\prime}) &= prep_{in}(s_2^{psv}, s^{\acute{}}_{nxt}). \end{aligned}$$

The induction hypothesis is

$$s_{nxt1} \vdash_{\mathsf{as}} s_1^{psv\prime} : s^{\acute{}}_{nxt} \quad \text{and} \quad s_{nxt2} \vdash_{\mathsf{as}} s_2^{psv\prime} : s^{\acute{}}_{nxt}.$$

Therefore, all three premises of Rule AS-If^p are satisfied.

Subcase $stmt = \texttt{while}(e)\ \{s_b^{psv}\}$

According to the definition of $prep_{in}$ it is

$$\begin{aligned} s^{`}_{nxt} &= \texttt{if}(e)\ \{s_{nxt1}\}\ \texttt{else}\ \{s^{\acute{}}_{nxt}\} \quad \text{and} \\ stmt' &= \texttt{while}(e)\ \{s_2^{psv}\} \quad \text{with} \\ (s_{nxt1}, s_1^{psv}) &= prep_{in}(s_b^{psv}, s^{\acute{}}_{nxt}) \quad \text{and} \\ (s_{nxt2}, s_2^{psv}) &= prep_{in}(s_1^{psv}, \texttt{if}(e)\ \{s_{nxt1}\}\ \texttt{else}\ \{s^{\acute{}}_{nxt}\}). \end{aligned}$$

The induction hypothesis is

$$s_{nxt2} \vdash_{\mathsf{as}} s_2^{psv} : \texttt{if}(e)\ \{s_{nxt1}\}\ \texttt{else}\ \{s^{\acute{}}_{nxt}\}.$$

Rule AS-While^p proves the claim.

Case $stmt = s^{act}$

Subcase $stmt = e!m(\overline{e})\{\overline{T_l}\ \overline{x_l};\ s^{psv};\ x = ?\texttt{return}(T\ x').\texttt{where}(e)\}$

According to the definition of $prep_{out}$ it is

$$\begin{aligned} stmt' &= s_{nxt};\ e!m(\overline{e})\{\overline{T_l}\ \overline{x_l};\ s^{psv\prime};\ [i]\,x = ?\texttt{return}(T\ x').\texttt{where}(e) \quad \text{with} \\ (s_{nxt}, s^{psv\prime}) &= prep_{in}(s^{psv}, next = i). \end{aligned}$$

Due to the induction hypothesis we know that $s_{nxt} \vdash_{\mathsf{as}} s^{psv\prime} : next = i$. This makes Rule AS-CallOut applicable which yields the proposition.

$\boxed{\textbf{Subcase}}$ $stmt = \texttt{while}(e)\ \{s^{act}\}$
According to the definition of $prep_{out}$ it is

$$stmt' \quad = \quad \texttt{while}(e)\ \{prep_{in}(s^{act})\},$$

so that the induction hypothesis directly implies the proposition.

□

The next lemma justifies the term anticipation-valid configuration.

Lemma C.2.5 (Dynamic anticipation-validity implies proper anticipation)**:** Assume a configuration $(h, \mathsf{v}, (\mu, mc)) \in Conf$, such that $h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc : s_{nxt}$. Then the following holds:

- If $\Delta \vdash (h, \mathsf{v}, (\mu, mc^{act}) \circ \mathsf{CS}) : \Theta \xrightarrow{\gamma!} \Delta \vdash (h, \mathsf{v}, (\mu, [i]\, mc^{psv}) \circ \mathsf{CS}) : \Theta'$ then $[\![next]\!]_h^{\mathsf{v},\mu} = i$.
- If $(h, \mathsf{v}, (\mu, mc^{psv}) \circ \mathsf{CS}) \rightsquigarrow^* (h, \mathsf{v}, (\mu, [i]\, mc^{psv\prime}) \circ \mathsf{CS})$ then $[\![next]\!]_h^{\mathsf{v},\mu} = i$.

Proof. Let us first assume that $h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc^{psv} : s_{nxt}$ and

$$(h, \mathsf{v}, (\mu, mc^{psv}) \circ \mathsf{CS}) \rightsquigarrow^* (h, \mathsf{v}, (\mu, [i]\, mc^{psv\prime}) \circ \mathsf{CS}).$$

If mc^{psv} starts with an instance of s^{psv} then the proposition immediately follows from the premises of Rule AD-s^{psv}. If mc^{psv} starts with an incoming return term then it follows immediately from the premise of Rule AD-RetI.

Now let us assume that

$$\Delta \vdash (h, \mathsf{v}, (\mu, mc^{act}) \circ \mathsf{CS}) : \Theta \xrightarrow{\gamma!} \Delta \vdash (h, \mathsf{v}, (\mu, [i]\, mc^{psv}) \circ \mathsf{CS}) : \Theta'.$$

If mc^{act} starts with an outgoing call or an outgoing return term, then the proposition follows from the passive case of this lemma. In all other cases it follows from the induction hypothesis. □

The last property that we have to show for proving Lemma 4.1.3 is that the dynamic anticipation-validity is an invariant regarding transitions of the operational semantics.

Lemma C.2.6 (Invariance of anticipation-validity)**:** Assume two specification language configurations, c and c', with

$$c = (h, \mathsf{v}, (\mu, mc)) \quad \text{such that} \quad h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc : s_{nxt}$$

and furthermore

$$c' = (h', \mathsf{v}', (\mu', mc')) \quad \text{with} \quad c \rightsquigarrow c' \quad \text{or} \quad \Delta \vdash c : \Theta \xrightarrow{a} \Delta' \vdash c' : \Theta'.$$

The it is also true that

$$h', \mathsf{v}', \mu' \vdash_{\mathsf{ad}} mc' : s_{nxt}.$$

Proof. Case analysis regarding the construction of mc of configuration c.

Case $mc = s^{psv};\ [i]?\texttt{return}(T\ x).\texttt{where}(e);\ mc^{act}$
We present three exemplary subcases, as the remaining cases are similar.

Subcase $s^{psv} = \mathsf{if}(e)\ \{s_1^{psv}\}\ \texttt{else}\ \{s_2^{psv}\};\ s_3^{psv}$
The assumed anticipation-validity regarding c is due to Rule AD-s^{psv}-RetI, which in particular implies

$$h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} \mathsf{if}(e)\ \{s_1^{psv}\}\ \texttt{else}\ \{s_2^{psv}\};\ s_3^{psv} : next = i. \tag{C.1}$$

According to the operational semantics, regarding c' we can conclude that $h' = h$, $\mathsf{v}' = \mathsf{v}$, and $\mu' = \mu$. Moreover, depending on the evaluation of e, the conditional statement reduces either to s_1^{psv} or s_2^{psv}. Without the loss of generality, let us assume that e evaluates to true. Thus,

$$c' = (h, \mathsf{v}, (\mu,\ s_1^{psv};\ s_3^{psv};\ [i]?\texttt{return}(T\ x).\texttt{where}(e);\ mc^{act})).$$

In order to prove $h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc' : s_{nxt}$, we have to show that

$$h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} s_1^{psv};\ s_3^{psv} : next = i.$$

Referring to Rule AD-s^{psv}, we can see from Equation C.1 that

$$_ \vdash_{\mathsf{as}} \mathsf{if}(e)\ \{s_1^{psv}\}\ \texttt{else}\ \{s_2^{psv}\};\ s_3^{psv} : next = i$$

which in turn implies that also

$$_ \vdash_{\mathsf{as}} s_1^{psv};\ s_3^{psv} : next = i$$

due to the Rules AS-If^p and AS-Seq^p. Furthermore the premise

$$(h, \mathsf{v}, (\mu, \mathsf{if}(e)\ \{s_1^{psv}\}\ \texttt{else}\ \{s_2^{psv}\};\ s_3^{psv})) \rightsquigarrow^* (h, \mathsf{v}, (\mu, [i]\ stmt_{in};\ s_4^{psv}))$$

of Rule AD-s^{psv} implies that also

$$(h, \mathsf{v}, (\mu, s_1^{psv}\ ;\ s_3^{psv})) \rightsquigarrow^* (h, \mathsf{v}, (\mu, [i]\ stmt_{in};\ s_4^{psv}))$$

is true. Therefore, we get

$$h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} s_1^{psv};\ s_3^{psv} : next = i$$

.

Subcase $s^{psv} = [j]\,(C\ x)?m(\overline{T}\ \overline{x}).\texttt{where}(e)\{\ \overline{T_l}\ \overline{x_l};\ s^{act};\ !\texttt{return}(e')\ \};\ s_3^{psv}$
Similar to the previous subcase, the premise of Rule AD-s^{psv}-RetI yields

$$h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} [j]\,(C\ x)?m(\overline{T}\ \overline{x}).\texttt{where}(e)\{\ \overline{T_l}\ \overline{x_l};\ s^{act};\ !\texttt{return}(e')\ \};\ s_3^{psv} : next = i,$$

which, according to the Rules AD-s^{psv}, AS-Seq^p, and AS-CallIn, implies that

$$[\![next]\!]_h^{\mathsf{v},\mu} = j \quad \text{and} \quad s^{act} = s_1^{act};\ s'_{nxt} \quad \text{with} \quad s'_{nxt} \vdash_{\mathsf{as}} s_3^{psv} : next = i.$$

The configuration c may only evolve to c' in terms of an incoming method call which leads to

$$c' = (h, \mathsf{v}, (\mathsf{v}_l \cdot \mu, s_1^{act};\ s'_{nxt}; !\texttt{return}(e');\ s_3^{psv};\ [i]?\texttt{return}(T\ x).\texttt{where}(e);\ mc^{act})).$$

Thus, it remains to show that

$$h, \mathsf{v}, \mathsf{v}_l \cdot \mu \vdash_{\mathsf{ad}} s_1^{act};\ s'_{nxt};\ !\texttt{return}(e');\ s_3^{psv} : next = i.$$

This, however, is true according to Rule AD-s^{act}-RetOut.

Subcase $s^{psv} = \epsilon$
In this subcase, the code mc of c starts with the outgoing call term $!\texttt{return}(e)$, so the assumption about the anticipation-validity regarding c is due to Rule Ad-RetI. Since its premise $_ \vdash_{\mathsf{as}} mc^{act} : s_{nxt}$ also implies anticipation-validity of mc^{act} regarding any heap and variable functions and since mc reduces to mc^{act} through an incoming return label, we can immediately see that

$$h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc^{act} : s_{nxt}.$$

Case $mc = s^{psv}$
As for configurations c whose code consist of a passive statement only, the corresponding proofs can be easily derived from the previous case. Basically, we only have to omit the trailing code $[i]\, x = ?\texttt{return}(T\ x).\texttt{where}(e);\ mc^{act}$.

Case $mc = s^{act};\ !\texttt{return}(e);\ mc^{psv}$
Also regarding active code, we will show the most interesting subcases.

Subcase $s^{act} = x{=}e;\ s_1^{act}$
Therefore, c internally reduces to

$$c' = (h, \mathsf{v}', (\mu', s_1^{act};\ !\texttt{return}(e);\ mc^{psv})).$$

According to Rule AD-s^{act}-RetOut it is $s_1^{act} = s_2^{act}; s'_{nxt}$ such that

$$s^{act} = x{=}e;\ s_2^{act};\ s'_{nxt} \quad \text{with} \quad s'_{nxt} \vdash_{\mathsf{as}} mc^{psv} : s_{nxt}.$$

We now have to distinguish the case, where x is the *next* variable, from the case where x represents a different variable.

Subsubcase $x \neq next$
As the first statement of s^{act} is not an outgoing call, but also not an instance of s_{nxt}, we know from Rule AD-s^{act} that

$$_ \vdash_{\mathsf{as}} x{=}e;\ s_2^{act};\ s'_{nxt} : s'_{nxt}.$$

Consequently, it is also true that

$$_ \vdash_{\mathsf{as}} s_2^{act};\ s'_{nxt} : s'_{nxt}.$$

This, in turn, leads to the fact that, according to Rule AD-s_{nxt}-RetOut, also

$$h, \mathsf{v}', \mu' \vdash_{\mathsf{ad}} s_2^{act};\ s'_{nxt};\ !\mathtt{return}(e);\ mc^{psv}$$

is true.

$\boxed{\textit{Subsubcase}}$ $x = next$

In this case the local variable list is not changed by the internal transition, i.e., $\mu' = \mu$. Moreover, we have to consult Rule AD-s_{nxt} instead of Rule AD-s^{act}. And this rule's two premises, applied to our assignment, leads to

$$(h, \mathsf{v}, (\mu, x = e)) \rightsquigarrow (h, \mathsf{v}', (\mu, \epsilon)),$$

such that $h, \mathsf{v}', \mu \vdash_{\mathsf{ad}} s_2^{act};\ s'_{nxt} : s'_{nxt}$. Therefore, in particular the first but also the second premise of Rule AD-s^{act}-RetOut are true regarding the configuration c'.

$\boxed{\textbf{Subcase}}$ $s^{act} = e!m(\overline{e})\ \{\overline{T}\ \overline{x};\ s^{psv};\ [i]\, x = ?\mathtt{return}(T\ x).\mathtt{where}(e')\ \};s_1^{act}$

Rule AD-$stmt_{out}$ yields

$$h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} s^{psv} : next = i \quad \text{and} \quad _ \vdash_{\mathsf{as}} s_1^{act} : s_{nxt}.$$

Thus, the transition from c to c' in terms of an outgoing method call label leads to

$$c' = (h, \mathsf{v}, (\mu, s^{psv};\ [i]\, x = ?\mathtt{return}(T\ x).\mathtt{where}(e');\ s_1^{act};\ !\mathtt{return}(e);\ mc^{psv})).$$

According to Rule Ad-s^{psv}-RetI, it remains to show that

$$_ \vdash_{\mathsf{as}} s_1^{act};\ !\mathtt{return}(e);\ mc^{psv} : s_{nxt}.$$

Since we assume that c is anticipation-valid and due to Rule AD-s^{act}-RetOut it is $s_1^{act} = s_2^{act}; s'_{nxt}$ such that

$$s'_{nxt} \vdash_{\mathsf{as}} mc^{psv} : s_{nxt}.$$

Therefore, according to Rule AS-s^{act}-RetOut, it is indeed

$$_ \vdash_{\mathsf{as}} s_2^{act};\ s'_{nxt};\ !\mathtt{return}(e);\ mc^{psv} : s_{nxt}.$$

$\boxed{\textbf{Subcase}}$ $s^{act} = \epsilon$

Therefore, it is

$$h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} !\mathtt{return}(e);\ mc^{psv} : s_{nxt}$$

and additionally

$$h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc^{psv} : s_{nxt}.$$

Since c evolves to

$$c' = (h, \mathsf{v}, (\mu, mc^{psv})),$$

this implies $h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc^{psv} : s_{nxt}$.

$\boxed{\boxed{\textbf{Case}}}$ $mc = s^{act}$

Much as the proof for passive statement represents a simplified case of passive call stack code, also the proof for active statements are very similar to the previous proof case. □

C.3 Correctness of the generated code

In this section we want to prove that a preprocessed specification and the correspondingly generated *Japl* code are testing bisimilar. This will also represent a proof for Lemma 3.6.2 as it stated for each specification the general existence of a program of the programming language which is "trace-equal" modulo input-enabledness. To prove testing bisimilarity, we will first define a binary relation R_t over specification language and programming language configurations. Afterwards we will prove that R_t is a testing bisimulation. Note in this section we have to deal with constructs of the specification language and, at the same time, with constructs of the programming language sharing the same name due to our language extension approach. Therefore, in the following, we will annotate constructs of the specification language with *sp* (e.g. $stmt_{sl}$) and those of the programming language with *pl* (e.g. $stmt_{pl}$). Yet we may omit the annotation in cases where the affiliation of a construct is clear.

The relation R_t is defined over configurations. However, the definition will be based on similar relations over statements and, respectively, over call stacks. Thus, before we will give the actual definition for R_t we need to define the relations regarding statements and call stacks.

Definition C.3.1: The relation $\sim_{st} \subseteq stmt_{sl} \times stmt_{pl}$ is recursively defined by the equations shown in Table C.3.

$$
\begin{array}{l}
s^{psv} \sim_{st} \varepsilon \\
\texttt{if}(e)\ \{s_1^{act}\}\ \texttt{else}\ \{s_2^{act}\} \sim_{st} \texttt{if}(e)\ \{stmt_1\}\ \texttt{else}\ \{stmt_2\} \\
\quad \text{with } s_1^{act} \sim_{st} stmt_1 \text{ and } s_2^{act} \sim_{st} stmt_2 \\
\texttt{while}(e)\ \{s^{act}\} \sim_{st} \texttt{while}(e)\ \{stmt\} \quad \text{with } s^{act} \sim_{st} stmt \\
s_1^{act};\ s_2^{act} \sim_{st} stmt_1;\ stmt_2 \quad \text{with } s_1^{act} \sim_{st} stmt_1 \text{ and } s_2^{act} \sim_{st} stmt_2 \\
x = e \sim_{st} x = e \\
e!m(\overline{e})\ \{\ s^{psv}; [i]\, x = ?\texttt{return}(T\ x).\texttt{where}(e)\ \} \sim_{st} x = e.m(\overline{e});\ check(i, e) \\
\texttt{new}!C(\overline{e})\{\ s^{psv}; [i]\, x = ?\texttt{return}(C\ x).\texttt{where}(e)\ \} \sim_{st} x = \texttt{new}\ C(\overline{e});\ check(i, e)
\end{array}
$$

Table C.3: Simulation relation for statements

Note, the relation $\sim_{st}$ relates all passive (specification language) statements to the empty (programming language) statement. Similarly, active method and constructor call statements of the specification language are related to the corresponding method or constructor call of the programming language, ignoring the passive statement s^{psv} that forms the body of the original call expectation statement.

Additionally, note that regarding the relation $\sim_{st}$, the expectation bodies of method and constructor calls must not provide variable declarations. Likewise, the block statement is not part of the relation. Therefore, a specification statement (as well as the corresponding program statement) of this relation never contains local variable declarations apart from the formal parameters of incoming calls.

Lemma C.3.2: Assume a preprocessed specification statement $stmt_{sl}$ and, correspondingly, a programming language statement $stmt_{pl}$ that results from generating code from $stmt_{sl}$ by means of $code_{in}$ or, respectively, $code_{out}$. Then it is $stmt_{sl} \sim_{st} stmt_{pl}$.

Proof. By structural induction. Straightforward. For instance, all passive statements are completely transcribed to *method body* code by $code_{in}$ such that no main body statement is generated at all. Similarly, all other cases immediately follow from the definition of $code_{out}$, given in Tale 4.5, and the definition of $\sim_{st}$, given in Table C.3. □

The next definition specifies a relation over activation records of the specification language and the corresponding call stack of the programming language. The definition is based on the previously defined relation over statements. However, it additionally has to consider the languages' different handling concerning the local variables. For, regarding the specification language, an incoming call results in an extension of the local variable list of the call stack's topmost (and only) activation record by a local variable functions v_l capturing the parameters of an incoming call. Within the programming language, in contrast, an incoming call causes the creation of a new activation record with its own variable function list. Moreover, while we assume that the specification does not introduce any local variables (apart from the parameter of a incoming method or constructor call), meaning that the local variable functions only consists of the formal parameters, the corresponding variable function of the programming language, in contrast, additionally provides a variable *retVal*.

Definition C.3.3: The relation $\sim_{CS} \subseteq \mathsf{AR}_{sl} \times \mathsf{CS}_{pl}$ consists of pairs of specification activation records and programming language calls stacks. It is $(\mu_{sl}, mc_{sl}) \sim_{CS} \mathsf{CS}_{pl}$ in exactly the following cases

1. $(\mathsf{v}_\perp, s^{act}) \sim_{CS} (\mathsf{v}_\perp, stmt;\ \mathtt{return})$ if $s^{act} \sim_{st} stmt$,
2. $(\mathsf{v}\cdot\mu, s^{act}; !\mathtt{return}(e);\ mc^{psv}) \sim_{CS} (\check{\mathsf{v}}, stmt;\ retVal = e;\ \mathtt{return}(retVal)) \circ \mathsf{CS}^{eb}$ if $s^{act} \sim_{st} stmt$ and $(\mu, mc^{psv}) \sim_{CS} \mathsf{CS}^{eb}$,
3. $(\mathsf{v}_\perp, s^{psv}) \sim_{CS} (\mathsf{v}_\perp, \varepsilon)$, and
4. $(\mathsf{v}_\perp\cdot\mathsf{v}\cdot\mu, s^{psv};\ [i]\,x = ?\mathtt{return}(T\ x).\mathtt{where}(e);\ mc^{act}) \sim_{CS} (\check{\mathsf{v}}, \mathtt{rcv}\ T{:}x;\ check(i,e);\ mc) \circ \mathsf{CS}'$ if $(\mathsf{v}\cdot\mu, mc^{act}) \sim_{CS} (\check{\mathsf{v}}, mc) \circ \mathsf{CS}'$.

With $\check{\mathsf{v}}$ we denote the variable function that results from extending v with an additional variable *retVal*.

Before we can define the actual testing bisimulation relation R_t we have to deal with another crucial difference between a specification and a program. That is, a program provides method code which is to be copied into the program configuration at runtime, whenever a corresponding method invocation occurs. Hence, relating configurations of the specification language with configurations of the

programming language is not sufficient but the static code, given in terms of method body code, has to meet certain requirements, as well. One solution would be to extend the codomain of the relation R_t such that it does not only comprise the configurations $Conf_{pl}$ of the programming language but additionally its programs p. Thus, the relation R_t would be a subset of $Conf_{sl} \times (p \times Conf_{pl})$. However, static code, as the name implies, does not change during the program execution. To express this, we choose a slightly different approach, that is, we annotate R_t^p with a specific program p, and *for each* p the relation R_t^p is a subset of $Conf_{sl} \times Conf_{pl}$. Based on this notation, we will now discuss the requirements of R_t^p that are related to the static code provided by p. This has the following three aspects.

- The program p on its own has to provide certain features which are independent of any configurations. In particular, if p does not have these features then the corresponding relation R_t^p is the empty set.

- It has been said, that static code may be copied into the program configuration in order to execute it. Executability entails the requirement that certain expressions within the code must be evaluable. This, in turn, imposes corresponding requirements on the configurations of R_t^p regarding the variable assignments given in terms of the configuration's variable functions . Thus, on the one hand, the variable functions of a configuration of R_t^p have to provide values of a proper type such that the expressions can be evaluated. On the other hand, the code of a configuration of R_t^p must not implement assignments to variables which result in a wrongly typed variable.

- Finally, the method code within p must be able to simulate all the incoming call expectations that are implemented in specification configuration of R_t^p.

In the following, we will discuss these three aspects in more detail and provide corresponding definitions. Afterwards, we will use these definitions to formulate the definition of the relation R_t^p.

First, let us deal with the general requirements regarding the static code itself. For instance, a straightforward requirement is that all methods must provide well-formed code, only. More specifically, as we have discussed in Chapter 4, the body of a method must provide a structure that allows the simulation of, not only one but potentially several, incoming call statements. To this end, we assume that the code structure follows our anticipation strategy, meaning that each method definition of p implements a case switch regarding the communication identifier and the corresponding where-clause. This requirement is formulated by the following definition.

Definition C.3.4 (Anticipation-based code structure)**:** Assume a well-typed program p. We say that p has an *anticipation-based code structure*, if for each method m of each

class C of p the definition is of the following form

$$T\ m(\overline{T}\,\overline{x})\{\ T\ retVal;$$
$$\prod_{k=1}^{n}\ (\mathsf{if}((next == i_k)\ \&\&\ (e_k))\ \{\ stmt_k;\ retVal = e'_k\ \}\ \texttt{else}\ )\ \{fail;\}$$
$$\texttt{return}(retVal)\ \}$$

and, correspondingly, for each class C the definition of its constructor is of the following form

$$C\ C(\overline{T}\,\overline{x})\{\ \mathsf{if}(internal)\ \{\varepsilon\}\ \texttt{else}$$
$$\prod_{k=1}^{n}\ (\mathsf{if}((next == i_k)\ \&\&\ (e_k))\ \{\ stmt_k\ \}\ \texttt{else}\ )\ \{fail;\}$$
$$\texttt{return}\ \}.$$

We use the $\prod$ symbol to denote an iteration of nested conditional statements. Each condition expression tests for the next expected communication identifier and the corresponding where-clause. If the method invocation does not match any implemented call expectations regarding this method, then *fail* is called.

As for the relation R_t^p we will presume that p has an anticipation-based code structure. Otherwise, the relation is considered to be the empty set. Note that this requirement can be checked independently of any configurations. If p has the desired structure, however, it imposes additional requirements on the configurations of R_t^p. On the one hand, it is necessary that for each method the expressions e_1 to e_n of Definition C.3.4 can be evaluated. Since we assume that the program does not use local variables or fields (cf. code generation algorithm), this represents a requirement on the global variable function v of the configurations. In particular, v must provide defined values for all global variables that occur in e_1 to e_n. Specifically, the types of the provided values must be as assumed by the expressions, as otherwise their evaluation is not defined and the program can get stuck. Moreover, the code of a specification configuration of R_t^p must not change the type of global variables by performing a wrongly typed assignment.

Definition C.3.5 (Well-typed variable function and specification configuration)**:** Let Δ be a global and Γ a local type mapping. Further, assume a variable function list $\mu = \mathsf{v}\cdot\mu'$. We say, μ is *well-typed* regarding Γ and Δ, written

$$\Gamma; \Delta \vdash_{\mathsf{var}} \mathsf{v}\cdot\mu' : \mathsf{ok},$$

if, and only if,

$$\begin{aligned}\Gamma = \Gamma_1, \Gamma_2 \quad \text{such that } & dom(\Gamma_1) = dom(\mathsf{v}),\\ & \text{for all } x \in dom(\mathsf{v}).\ \Delta(\mathsf{v}(x)) = \Gamma_1(x), \text{ and}\\ & \Gamma_2; \Delta \vdash_{\mathsf{var}} \mu' : \mathsf{ok}.\end{aligned}$$

$$[\text{T-}s^{act}\text{-RETO}]\frac{\Gamma;\Delta \vdash \mathit{stmt} : \mathsf{ok}^{act} \qquad \Gamma;\Delta \vdash mc^{psv} : \mathsf{ok}^{psv}}{\Gamma;\Delta \vdash \mathit{stmt};\ !\mathtt{return}(e);\ mc^{psv} : \mathsf{ok}^{psv}}$$

$$[\text{T-}s^{psv}\text{-RETI}]\frac{\Gamma;\Delta \vdash \mathit{stmt} : \mathsf{ok}^{psv} \qquad \Gamma;\Delta \vdash mc^{act} : \mathsf{ok}^{act}}{\Gamma;\Delta \vdash \mathit{stmt};\ ?\mathtt{return}(T\ ).\mathtt{where}(e);\ mc^{act} : \mathsf{ok}^{psv}}$$

Table C.4: Well-typedness of dynamic specification code mc_{sl}

Moreover, for a configuration $c_{sl} = (h, \mathsf{v}, (\mu, mc))$ of the specification language, we say that c_{sl} is *well-typed* regarding Γ and Δ, written

$$\Gamma;\Delta \vdash_{\mathsf{var}} c_{sl} : \mathsf{ok},$$

if

$$\Gamma;\Delta \vdash_{\mathsf{var}} \mathsf{v}\cdot\mu : \mathsf{ok} \quad \text{and if the judgment} \quad \Gamma;\Delta \vdash mc : \mathsf{ok}^{\gamma}$$

is derivable regarding the inference rules given in Table 3.2 and Table C.4.

While we have just seen that a configuration has to provide certain features, such that the method bodies of p can be executed properly, we still have to formulate the requirement that, contrary, p indeed provides method code that matches the expectations specified within the configuration specification. In particular, the code provided by p has to match the expectations in such a way that for each incoming call statement regarding method m within the configuration specification, we can find corresponding code in the method definition of m within p. This requirement is defined as follows.

Definition C.3.6 (Expectation supporting code)**:** Let mc_{sl} be activation record code regarding the specification language which is annotated with expectation ids. A program p with anticipation-based code structure *supports all expectations* of mc_{sl}, written

$$p \rhd mc_{sl},$$

if

- for each

  $$\big([i]\,(C\ x)?m(\overline{T}\ \overline{x}).\mathtt{where}(e)\ \{\mathit{stmt}_{sl};\ \mathtt{return}(e_r)\ \}\big)\ \in mc_{sl},$$

  there exist a corresponding conditional branch in the method definition of m in p such that

  $$\big(\mathsf{if}((\mathit{next} == i)\ \&\&\ (e))\ \{\ \mathit{stmt}_{pl};\ \mathit{retVal} = e_r\ \}\ \mathtt{else}\ \mathit{stmt}'_{pl}\big)\ \in p.C.m$$

  with $\mathit{stmt}_{sl} \sim_{st} \mathit{stmt}_{pl}$.

- for each

  $$\big([i]\,\mathtt{new}(C\ x)?C(\overline{T}\ \overline{x}).\mathtt{where}(e)\ \{\mathit{stmt}_{sl};\ \mathtt{return}\}\big)\ \in mc_{sl},$$

there exist a corresponding conditional branch in the constructor definition of C in p such that

$$\big(\mathsf{if}((next == i)\ \&\&\ (e))\ \{\ stmt_{pl}\}\ \texttt{else}\ stmt'_{pl}\big)\ \in p.C.m$$

with $stmt_{sl} \sim_{st} stmt_{pl}$.

Moreover, each expectation identifier that occurs within a conditional branch of a method or a constructor definition is unique.

Finally, we can define the relation R_t^p.

Definition C.3.7 (Testing bisimulation relation R_t^p)**:** Assume a program p with an anticipation-based code structure. Further, assume a type mapping Δ such that for all methods m of all classes C in p and for all Boolean expression e_1 to e_n of m according to Definition C.3.4 it is

$$\Gamma_g, \Gamma_{C.m}; \Delta \vdash e_k : \mathsf{Bool},$$

where $\Gamma_{C.m}$ represents the local type mapping due to the formal parameters and local variables of $C.m$ according to Rule T-MDEF in Table 2.2 and Γ_g is the local type mapping that results from p's global variables according to Rule T-PROG'.

We define a relation $R_t^b \subseteq Conf_{sl} \times Conf_{pl}$ over configurations of the specification language and of the programming language as follows. For all heap functions h and all global variable functions v the relation R_t^p exactly consists of the following pairs: It is

$$((h, \mathsf{v}, \mathsf{CS}_{sl}), (h, \mathsf{v}, \mathsf{CS}_{pl})) \in R_t$$

if, and only if,

1. regarding the call stacks it is

$$\mathsf{CS}_{sl} = (\mu, mc_{sl}) \quad \text{and} \quad (\mu, mc_{sl}) \sim_{CS} \mathsf{CS}_{pl},$$

2. the program p supports all expectations of mc_{sl}, i.e.,

$$p \rhd mc_{sl},$$

3. the specification configuration is well-typed regarding the local type mapping Γ_g and the global type mapping Δ of p, i.e.,

$$\Gamma_g; \Delta \vdash_{\mathsf{var}} (h, \mathsf{v}, (\mu, mc_{sl}) : \mathsf{ok}.$$

and

4. the specification configuration is anticipation-valid, i.e.,

$$h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc_{sl} : \mathsf{anticip}$$

Note, the heap and the global variables of related configurations are identical. Moreover, the call stack of the specification's configuration consists of a single activation record, only, and it must be related to the call stack of the program's configuration in terms of the relation $\sim_{CS}$.

Note further that, according to the operational semantics of the specification language, the call stack of a specification's configuration always consists of only one activation record. Hence, the corresponding equation, $\mathsf{CS}_{sl} = \mathsf{AR}_{sl}$ in Definition C.3.7 does not represent a real restriction.

Now, the following lemma will show that the relation R_t^p is a testing bisimulation as defined in 4.4.6. To understand the structure of the lemma's proof, recall that the code mc of a configuration's activation record is always either active, mc^{act}, or passive, mc^{psv}, code. In particular, it is always of the following form:

$$\begin{array}{lcl} mc^{act} & ::= & s^{act} \mid s^{act};\ !\texttt{return}(e);\ mc^{psv} \\ mc^{psv} & ::= & s^{psv} \mid s^{psv};\ x = ?\texttt{return}(T\ x).\texttt{where}(e);\ mc^{act} \end{array}$$

That is, the code of an activation record either consists of single statement (s^{act} or s^{psv}, respectively) or it consists of a statement followed by a return term and some more activation record code mc^{psv} or mc^{act}.

The proof of the lemma consists of a case analysis regarding the construction of the specification configurations of the relation R_t^p.

Lemma C.3.8: The binary relation R_t^p, defined in C.3.7, indeed represents a testing bisimulation as defined in 4.4.6.

Proof. Assume a program p with anticipation-based code structure. Further, assume a specification language configuration c_{sl} and a programming language specification c_{pl}, such that

$$(c_{sl}, c_{pl}) \in R_t^p. \tag{Ass}$$

The definition of R_t^p implies that there exist a heap function h, a global variable function v, as well as an activation record of the specification language $\mathsf{AR}_{sl} = (\mu, mc_{sl})$ and a call stack of the programming language CS_{pl} such that

$$c_{sl} = (h, \mathsf{v}, \mathsf{AR}_{sl}) \quad \text{and} \quad c_{pl} = (h, \mathsf{v}, \mathsf{CS}_{pl}) \quad \text{with } \mathsf{AR}_{sl} \sim_{CS} \mathsf{CS}_{pl}.$$

Similar to the proof of Lemma C.1.3, we make a *case analysis* regarding the construction of the code mc_{sl} of AR_{sl}. For each case we will prove that c_{pl} simulates c_{sl} ($\Rightarrow$) and additionally that c_{sl} simulates c_{pl} up to test faults ($\Leftarrow$). Specifically, we have to show *for each case* that the two configurations allow for similar computations steps where the resulting configurations, c'_{sl} and c'_{pl}, again meet the four requirements of Definition C.3.7. Two of the four requirements, however, can be shown generally without analyzing distinct cases. For, we have already shown in Lemma C.2.6 that anticipation validity is invariant concerning computation steps

of the operational semantics. Moreover, it is obvious that, if p supports all expectations that are specified in c_{sl} then no computation step adds new expectations, so that p also supports all expectations specified in the new configuration c'_{sl}.

As for the following case analysis, we first consider the cases, where AR_{sl} contains active code mc^{act}. Afterwards, we consider all cases, where the code of AR_{sl} is passive, hence, an instance of mc^{psv}.

Case $\mathsf{AR}_{sl} = (\mathsf{v}_l{\cdot}\mu',\ s^{act};\ !\mathtt{return}(e);\ mc^{psv})$ with $s^{act} \neq \epsilon$
Thus, the configurations c_{sl} is of the following form

$$c_{sl} = (h, \mathsf{v}, (\mathsf{v}_l{\cdot}\mu',\ s^{act};\ !\mathtt{return}(e);\ mc^{psv})).$$

In particular, it is $\mu = \mathsf{v}_l{\cdot}\mu'$. So, according to Definition C.3.7 as well as Definition C.3.3, we know from (Ass) that

$$c_{pl} = (h, \mathsf{v}, (\check{\mathsf{v}}_l,\ stmt;\ retVal = e;\ \mathtt{return}(retVal)) \circ \mathsf{CS}^{eb}),$$

such that

$$s^{act} \sim_{st} stmt \quad \text{and} \quad (\mu', mc^{psv}) \sim_{CS} \mathsf{CS}^{eb}.$$

We make a subcase analysis regarding the first active statement of s^{act}.

Subcase $s^{act} = x{=}e;\ s_1^{act}$
Then $s^{act} \sim_{st} stmt$ implies that

$$stmt = x{=}e;\ stmt_1 \quad \text{with } (*)\ s_1^{act} \sim_{st} stmt_1.$$

Direction $\Rightarrow$
According to the operational semantics of the specification language, c_{sl} may reduce to c'_{sl} only in terms of an internal computation step such that

$$c_{sl} \rightsquigarrow c'_{sl} = (h, \mathsf{v}', (\mathsf{v}_l{\cdot}\mu',\ s_1^{act};\ !\mathtt{return}(e);\ mc^{psv}))$$

Note that the local variables did not change as (Ass) implies that x is not a local variable or parameter. Thus, similarly, we have

$$c_{pl} \rightsquigarrow c'_{pl} = (h, \mathsf{v}', (\check{\mathsf{v}}_l,\ stmt_1;\ retVal = e;\ \mathtt{return}(retVal)) \circ \mathsf{CS}^{eb}).$$

So due to (Ass) and $(*)$ it is

$$(\mathsf{v}_l{\cdot}\mu',\ s_1^{act};\ !\mathtt{return}(e);\ mc^{psv}) \sim_{CS} (\check{\mathsf{v}}_l,\ stmt_1;\ retVal = e;\ \mathtt{return}(retVal)) \circ \mathsf{CS}^{eb}).$$

Again, the assumption (Ass) and Rule T-Seq of Table 2.2 imply that

$$\Gamma_g; \Delta \vdash_{\mathsf{var}} (h, \mathsf{v}', (\mathsf{v}_l{\cdot}\mu', s_1^{act};\ !\mathtt{return}(e);\ mc^{psv})) : \mathsf{ok}.$$

Thus, according to Definition C.3.7 we get

$$(c'_{sl}, c'_{pl}) \in R_t.$$

Direction $\Leftarrow$
The variable x must not be the extra variable *retVal*. Furthermore, c_{pl} can only deterministically reduce to the above mentioned c'_{pl}. Hence, this proof direction results in the same configuration pair

$$(c'_{sl}, c'_{pl}) \in R_t.$$

Subcase $s^{act} = e_c!m(\overline{e})\{s^{psv};\ [i]\,x = ?\texttt{return}(T\ x').\texttt{where}(e')\};\ s_1^{act}$
In particular due to Definition C.3.1, the assumption (Ass) implies

$$c_{pl} = (h, \mathsf{v}, (\check{\mathsf{v}}_l, e_c.m(\overline{e});\ stmt_1;\ retVal = e;\ \texttt{return}(retVal)) \circ \mathsf{CS}^{eb}).$$

Direction $\Rightarrow$
Configuration c_{sl} reduces to c'_{sl} due to an outgoing method call. Hence,

$$\Delta \vdash c_{sl} : \Theta \xrightarrow{a} \Delta \vdash c'_{sl} : \Theta',$$

with

$$a = \nu(\Theta').\langle call\ o.m(\overline{v})\rangle! \quad \text{such that} \quad o = [\![e_c]\!]_h^{\mathsf{v},\mu} \ \text{ and } \ \overline{v} = [\![\overline{e}]\!]_h^{\mathsf{v},\mu}.$$

and

$$c'_{sl} = (h, \mathsf{v}, (\mathsf{v}_\perp \cdot \mu,\ s^{psv};\ [i]\,x = ?\texttt{return}(Tx').\texttt{where}(e');\ s_1^{act};\ !\texttt{return}(e);\ mc^{psv})).$$

In the following, let us refer to the code of c'_{sl} by mc'_{sl}. Note that the new local variable function is the completely undefined variable function $\mathsf{v}_\perp$, since the code of c_{sl} is free of local variable declarations.

As for the programming language configuration c_{pl}, the topmost statement of the topmost activation record is the outgoing call $e_c.m(\overline{e})$ which likewise leads to a transition labeled with the same communication label a, such that

$$\Delta \vdash c_{pl} : \Theta \xrightarrow{a} \Delta \vdash c'_{pl} : \Theta',$$

with

$$c'_{pl} = (h, \mathsf{v}, (\check{\mathsf{v}}_l, \texttt{rcv}\ x{:}T;\ stmt_1;\ retVal = e;\ \texttt{return}(retVal)) \circ \mathsf{CS}^{eb}).$$

In the following, let us refer to the code of c'_{pl} by mc'_{pl}. According to (Ass) and Definition C.3.1, it is

$$(\mathsf{v}_\perp \cdot \mu,\ mc'_{sl}) \sim_{st} ((\check{\mathsf{v}}_l, \texttt{rcv}\ x{:}T;\ stmt_1;\ retVal = e;\ \texttt{return}(retVal)) \circ \mathsf{CS}^{eb}).$$

Furthermore, Rule T-CallOut of Table 3.2 and Rule T-s^{psv}-RetI of Table C.4 imply that

$$\Gamma_g; \Delta \vdash_{\mathsf{var}} (h, \mathsf{v}, (\mathsf{v}_\perp \cdot \mu,\ mc'_{sl})) : \mathsf{ok}.$$

Hence, it is

$$(c'_{sl}, c'_{pl}) \in R_t.$$

Direction $\Leftarrow$
Similar to the previous subcase, the configuration c_{pl} allows at most the same labeled transition to the configuration c'_{pl} that was introduced in the above proof regarding the other implication direction. This results in the same configuration pair such that, again,

$$(c'_{sl}, c'_{pl}) \in R_t.$$

The other subcases are similar.

Case $\mathsf{AR}_{sl} = (\mathsf{v}_l{\cdot}\mu',\ !\mathtt{return}(e);\ mc^{psv})$
Referring to Definition C.3.3, we can derive from (Ass), that

$$c_{sl} = (h, \mathsf{v}, (\mathsf{v}_l{\cdot}\mu', !\mathtt{return}(e);\ mc^{psv}))$$

and, on the other hand, that

$$c_{pl} = (h, \mathsf{v}, (\check{\mathsf{v}}_l, retVal = e; \mathtt{return}(retVal)) \circ \mathsf{CS}^{eb})$$

or

$$c_{pl} = (h, \mathsf{v}, (\check{\mathsf{v}}_l, \mathtt{return}(retVal)) \circ \mathsf{CS}^{eb}),$$

where we additionally know in the latter case that $\check{\mathsf{v}}_l(retVal) = [\![e]\!]_h^{\mathsf{v},\mu}$. Moreover, we know that

$$(\mu', mc^{psv}) \sim_{CS} \mathsf{CS}^{eb}.$$

Direction $\Rightarrow$
The only transition that may originate from c_{sl} is the one that is labeled with an outgoing return label a such that

$$a = \nu(\Theta').\langle return(v)\rangle! \quad \text{with } v = [\![e]\!]_h^{\mathsf{v},\mu}.$$

More specifically, due to Rule RetO of Table 3.3 we get

$$\Delta \vdash c_{sl} : \Theta \xrightarrow{a} \Delta \vdash c'_{sl} : \Theta' \quad \text{with } c'_{sl} = (h, \mathsf{v}, (\mu', mc^{psv})).$$

It is easy to see that processing the programming language configuration c_{pl} leads to the same outgoing communication step – with an intermediate internal computation step, if the case may be. In particular, in both cases, it is $\check{\mathsf{v}}_l(retVal) = [\![e]\!]_h^{\mathsf{v},\mathsf{v}_l{\cdot}\mu'}$ right before the outgoing return is processed. Therefore, it is

$$\Delta \vdash c_{pl} : \Theta \Longrightarrow \Delta \vdash c'_{pl} : \Theta' \quad \text{with } c'_{pl} = (h, \mathsf{v}, \mathsf{CS}^{eb}).$$

The assumption (Ass) immediately yields that

$$(\mu', mc^{psv}) \sim_{CS} \mathsf{CS}^{eb}.$$

Well-typedness of c'_{sl} results from Rule T-s^{act}-RetOut such that

$$\Gamma_g; \Delta \vdash_{\mathsf{var}} c'_{sl} : \mathsf{ok}.$$

So, all in all we can infer that

$$(c'_{sl}, c'_{pl}) \in R_t.$$

Direction $\Leftarrow$
Again, c_{pl} deterministically evolves to the configuration c'_{pl} of the previous proof direction.
Case $\mathsf{AR}_{sl} = (\mathsf{v}_l,\ s^{act})$
The proof of this case is almost identical to the previous two proof cases. Specifically, we only have to skip the proof obligation that the trailing call stack CS^{eb} relates to the corresponding specification code, as no trailing call stack exists in this case.
Case $\mathsf{AR}_{sl} = (\mu,\ s^{psv};\ [i]\, x = ?\mathtt{return}(T\, x').\mathtt{where}(e');\ mc^{act})$
Due to Definition C.3.3, it is $\mu = \mathsf{v}_\perp \cdot \mathsf{v}_l \cdot \mu'$ so that

$$c_{sl} = (h, \mathsf{v}, (\mathsf{v}_\perp \cdot \mathsf{v}_l \cdot \mu',\ s^{psv};\ [i]\, x = ?\mathtt{return}(T\, x').\mathtt{where}(e');\ mc^{act}).$$

Moreover the same definition leads to

$$c_{pl} = (h, \mathsf{v}, (\check{\mathsf{v}}_l, \mathtt{rcv}\ x{:}T;\ check(i, e');\ mc) \circ \mathsf{CS}^{eb}) \quad \text{with}$$
$$(\mathsf{v}_l, mc^{act}) \sim_{CS} (\check{\mathsf{v}}_l,\ mc) \circ \mathsf{CS}^{eb}.$$

We consider some subcases regarding the structure of s^{psv}. However, this time we will not consider both implication directions for each subcase but only the simulation direction ($\Rightarrow$). We will prove the simulation-up-to-faults direction ($\Leftarrow$) for all subcases at the end.
Subcase $s^{psv} = \mathsf{if}\ (e)\ \{s_1^{psv}\}\ \mathtt{else}\ \{s_2^{psv}\};\ s_3^{psv}$
Without loss of generality we can assume that $[\![e]\!]_h^{\mathsf{v},\mu} = \mathit{true}$ and thus

$$c_{sl} \rightsquigarrow c'_{sl} \quad \text{with} \quad c'_{sl} = (h, \mathsf{v}, (\mu,\ s_1^{psv};\ s_3^{psv};\ [i]\, x = ?\mathtt{return}(Tx').\mathtt{where}(e');\ mc^{act})).$$

However, again due to Definition C.3.3 it is

$$(\mu,\ s_1^{psv};\ s_3^{psv};\ [i]\, x = ?\mathtt{return}(T\, x').\mathtt{where}(e');\ mc^{act}) \sim_{CS} \mathsf{CS}_{pl}.$$

Due to Rule T-s^{act}-RetOut of Table C.4 and due to Rule T-Cond and Rule T-Seq of Table 3.2 we know that

$$\Gamma_g; \Delta \vdash_{\mathsf{var}} (\mu,\ s_1^{psv};\ s_3^{psv};\ [i]\, x = ?\mathtt{return}(T\, x').\mathtt{where}(e');\ mc^{act}) : \mathsf{ok}.$$

Thus, we get

$$(c'_{sl}, c_{pl}) \in R_t.$$

$\boxed{\textbf{Subcase}}$ $s^{psv} = [j]\,(C\ x)?m(\overline{T}\,\overline{x}).\texttt{where}(e')\{\ s^{act};\ \texttt{return}(e_r)\ \};\ s_3^{psv}$
In this case c_{sl} may only evolve due to an appropriate incoming method call label. That is,

$$\Delta \vdash c_{sl} : \Theta \xrightarrow{a} \Delta' \vdash c'_{sl} : \Theta,$$

with

$$c'_{sl} = (h, \mathsf{v}, (\mathsf{v}'_l{\cdot}\mu,\ s^{act};\ !\texttt{return}(e_r)\ ;\ s_3^{psv};\ [i]\,x = ?\texttt{return}(Tx').\texttt{where}(e');\ mc^{act}))$$

as well as

$$a = \nu(\Theta').\langle call\ o.m(\overline{v})\rangle? \quad \text{such that} \quad \Delta, \Delta', \Theta \vdash o, \overline{v} : C, \overline{T} \ \text{ and } \ [\![e']\!]_h^{\mathsf{v},\mathsf{v}'_l\cdot\mu}.$$

Let us refer to the code of c'_{sl} as mc'_{sl}. The assumption $h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc_{sl} : \mathsf{anticip}$ implies that

$$(*)\ [\![next]\!]_h^{\mathsf{v},\mu} = j$$

due to Lemma C.2.6. As for the configuration c_{pl}, the facts that p provides an anticipation-based code structure and, in particular, that $p \rhd mc_{sl}$, and finally that the program is generally input enabled, lead to

$$\Delta \vdash c_{pl} : \Theta \xrightarrow{a}_p \Delta' \vdash c'_{pl} : \Theta,$$

with

$$c'_{pl} = (h, \mathsf{v}, (\check{\mathsf{v}}'_l,\ stmt;\ \texttt{return}(retVal)) \circ (\check{\mathsf{v}}_l,\ \texttt{rcv}\ x{:}T;\ \ mc) \circ \mathsf{CS}^{eb}).$$

such that, due to $(*)$, it is $c'_{pl} \rightsquigarrow^* c''_{pl}$ with

$$c''_{pl} = (h, \mathsf{v}, (\check{\mathsf{v}}'_l,\ stmt_1;\ retVal = e_r;\ \texttt{return}(retVal)) \circ (\check{\mathsf{v}}_l,\ \texttt{rcv}\ x{:}T;\ mc) \circ \mathsf{CS}^{eb})$$

and with

$$s^{act} \sim_{st} stmt_1.$$

Let us refer to the code of the topmost activation record of c''_{pl} as mc''_{pl}. Then it is

$$(\mathsf{v}'_l{\cdot}\mu, mc'_{sl}) \sim_{st} (\check{\mathsf{v}}'_l, mc''_{pl}) \circ \mathsf{CS}^{eb}.$$

Due to Rule T-s^{psv}-RetI and Rule T-CallIn it is

$$\Gamma_g; \Delta \vdash_{\mathsf{var}} c'_{sl} : \mathsf{ok}$$

and finally we get

$$(c'_{sl}, c''_{pl}) \in R_t.$$

$\boxed{\textbf{Direction}}$ $\Leftarrow$
As mentioned above, the call stack CS_{pl} of the program configuration c_{pl} is externally blocked. Thus, it may only evolve due to an incoming call or due to an incoming return. That is, we can assume that

$$\Delta \vdash c_{pl} : \Theta \xrightarrow{a}_p \Delta' \vdash c'_{pl} : \Theta.$$

And regarding the communication label a we have to differentiate two subcases.
$\boxed{\textbf{Subcase}}$ $a = \nu(\Delta_n).\langle call\ o.m(\overline{v})\rangle?$
Due to the anticipation-based code structure of p, the configuration c'_{pl} is of the following form:

$$c'_{pl} = (h, \mathsf{v}, (\check{\mathsf{v}}_l, stmt;\ \texttt{return}(retVal)) \circ \mathsf{CS}_{pl}),$$

where *stmt* implements a case switch regarding expectation ids in form of a nesting of conditional statements as described in Definition C.3.4. Assume that

$$(*)\ \mathsf{v}(next) = j.$$

$\boxed{Subsubcase}$ $\texttt{if}\ ((next == j)\&\&(e_j))\ \{stmt_j;\ retVal = e'_j\}\ \texttt{else}\ \{stmt'\}\ \in stmt$
Due to fact that p supports all expectations of the code of c_{sl}, i.e.,

$$p \rhd s^{psv};\ [i]\, x = ?\texttt{return}(T\, x').\texttt{where}(e');\ mc^{act},$$

we can infer that $j \neq i$. Moreover, (Ass) implies that

$$h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc_{sl} : \mathsf{anticip}$$

so Lemma C.2.6 and $(*)$ yield that

$$\begin{aligned} & c_{sl} \rightsquigarrow^* c'_{sl} \quad \text{with} \\ & c'_{sl} = (h, \mathsf{v}, (\mu,\ [j]\, stmt_{in};\ s_1^{psv}; \\ & \qquad\qquad [i]\, x = ?\texttt{return}(T\, x').\texttt{where}(e');\ mc^{act})). \end{aligned}$$

Again, since p supports all expectations of c_{sl}, it is indeed

$$stmt_{in} = (C\, x)?m(\overline{T}\, \overline{x}).\texttt{where}(e_j)\{\ s^{act};\ !\texttt{return}(e'_j)\ \}\ .$$

If $[\![e_j]\!]_h^{\mathsf{v},\mu_l\cdot\mu} = false$ then $\Delta \vdash c'_{sl} : \Delta \not\xrightarrow{a}$. But in this case also the corresponding conditional branch of m within p is evaluated to false such that the method reports a failure.

So let us assume that $[\![e_j]\!]_h^{\mathsf{v},\mu_l\cdot\mu} = true$. Then we get

$$\Delta \vdash c'_{sl} : \Delta \xrightarrow{a} \Delta' \vdash c''_{sl} : \Theta$$

with

$$c''_{sl} = (h, \mathsf{v}, (\mathsf{v}_l\cdot\mu, s^{act}; !\texttt{return}(e_j);\ s_1^{psv};\ ?\texttt{return}(T\, x').\texttt{where}(e');\ mc^{act})).$$

Let us refer to the activation record of c''_{sl} as AR''_{sl}. On the other hand, the program configuration c'_{pl} reduces to

$$c'_{pl} \rightsquigarrow^* c''_{pl} = (h, \mathsf{v}, (\check{\mathsf{v}}_l, stmt_j;\ retVal = e'_j;\ \texttt{return}(retVal)) \circ \mathsf{CS}_{pl}),$$

where, yet again due to the expectation support, it is

$$(**)\quad s^{act} \sim_{st} stmt_j.$$

Let us refer to the call stack of c''_{pl} as CS''_{pl}, then we get from (Ass) and from $(**)$ that

$$\mathsf{AR}''_{sl} \sim_{CS} \mathsf{CS}''_{pl}.$$

$\boxed{Subsubcase}$ $\texttt{if}\ ((next == j)\&\&(e_j))\ \{stmt_j;\ retVal = e'_j\}\ \texttt{else}\ \{stmt'\} \notin stmt$
That is, the method m does not provide a conditional branch regarding the communication identifier j. According to the structure of the method, this results in a failure report. Thus, we have to show that the specification configuration cannot realize an incoming call regarding a. Indeed, since $h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc_{sl} : \mathsf{anticip}$, we know from Lemma C.2.6 and from $(*)$ that

$$\Delta \vdash c_{sl} : \Theta \not\xrightarrow{a} .$$

$\boxed{\textbf{Subcase}}$ $a = \nu(\Delta_n).\langle return(v)\rangle?$
According to the operational semantics and due to the form of c_{pl} it is

$$\Delta, \Delta_n \vdash v{:}T$$

so that

$$\Delta \vdash c_{pl} : \Theta \xrightarrow{a} \Delta, \Delta_n \vdash c'_{pl} : \Theta$$

with $c'_{pl} = (h, \mathsf{v}', (\check{\mathsf{v}}_l,\ check(i, e');\ mc) \circ \mathsf{CS}^{eb})$. Since we assume that $check(i, e')$ tests whether $next = i$ and e' evaluates to true, we can differentiate two subsubcases.

$\boxed{Subsubcase}$ $[\![next == i]\!]_h^{\mathsf{v},\check{\mathsf{v}}_l} \wedge [\![e']\!]_h^{\mathsf{v},\check{\mathsf{v}}_l} = true$
In this case we can assume that

$$c'_{pl} \rightsquigarrow^* c''_{pl} = (h, \mathsf{v}', (\check{\mathsf{v}}_l,\ mc) \circ \mathsf{CS}^{eb}),$$

but also we know from $h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc_{sl} : \mathsf{anticip}$ that $s^{psv} = \epsilon$ and thus

$$\Delta \vdash c_{sl} : \Theta \xrightarrow{a} \Delta, \Delta' \vdash c'_{sl} : \Theta$$

with

$$c'_{sl} = (h, \mathsf{v}', (\mathsf{v}_l \cdot \mu', mc^{act})).$$

Finally, both,

$$(\mathsf{v}_l \cdot \mu', mc^{act}) \sim_{CS} (\check{\mathsf{v}}_l,\ mc) \circ \mathsf{CS}^{eb}$$

as well as

$$h, \mathsf{v}', \mathsf{v}_l \cdot \mu' \vdash_{\mathsf{var}} mc^{act} : \mathsf{ok}$$

immediately follow from (Ass).

$\boxed{\textit{Subsubcase}}$ $[\![next == i]\!]_h^{\mathsf{v},\check{\mathsf{v}}_l} \wedge [\![e']\!]_h^{\mathsf{v},\check{\mathsf{v}}_l} = \mathit{false}$
In this case, we assume that $check(i, e')$ reports a failure. The specification configuration, however, does not accept such an incoming return label a, hence,

$$\Delta \vdash c_{sl} : \Theta \not\xrightarrow{a} .$$

$\boxed{\textbf{Case}}$ $\mathsf{AR}_{sl} = (\mathsf{v}_l,\ s^{psv})$
Similar to the s^{act} case, this s^{psv} case, again, represents a simplified version of the previous case, as we can replay its proofs while omitting the proof obligations regarding the trailing call stack CS^{eb} and, respectively, mc^{act}. □

In order to finally prove the correctness of the code generation algorithm, we have to show that the initial configurations of a specification s and the initial configuration of the correspondingly generated test program p represent a pair of the testing bisimulation relation R_t^p.

Lemma C.3.9 (Correctness of the test code generation)**:** Assume a well-typed specification s. Moreover, let $s' = prep(s)$ be the specification that results from preprocessing s as defined in Definition 4.1.4 and let p be the correspondingly generated program according to the algorithm described in Section 4.3. If the main statement of s' is an active statement then

$$(c_{init}(s'), c_{init}(p)) \in R_t^p.$$

Otherwise it is

$$(c_{init}(s'), \overline{c_{init}}(p)) \in R_t^p.$$

In particular, it is $R_t^p \neq \emptyset$.

Proof. Assume a well-typed configuration

$$s = \overline{cutdecl}\ \overline{T}\ \overline{x};\ \overline{mokdecl}\ \{stmt\}.$$

Let $s' = prep(s)$. Then, according to Definition 4.1.4 we have

$$s = \overline{cutdecl}\ \overline{T}\ \overline{x};\ \overline{T'}\ \overline{x'};\ T\ next;\ \overline{mokdecl}\ \{stmt'\},$$

where $stmt'$ results

1. from enriching $stmt$ with anticipation code by means of the code processing functions $prep_{in}$ and $prep_{out}$ and

2. from "globalizing" all local variables within *stmt*, meaning that each variable declaration and formal parameter within *stmt* has a global counterpart in $\overline{x'}$ such that $stmt'$ is free of local variable declarations (apart from formal parameters). Moreover, all occurrences of local variables and parameters within *stmt* are replaced by the corresponding global counterpart.

It is easy to see that well-typedness of s implies well-typedness of s', hence, let us assume that $\Delta \vdash s' : \Theta$. Further let us assume that p with

$$p = \overline{impdecl};\ \overline{T}\,\overline{x};\ \overline{T'}\,\overline{x'};\ T\ next;\ \overline{cldef};\ \{stmt_{pl};\ \texttt{return}\}$$

is the test program generated from s' as described in Section 4.3. According to the code generation algorithm, the class definitions $\overline{impldecl}$ are generated by means of the code generation functions $code_{in}$ and $code_{out}$. From, the definitions of these functions, given in Table 4.5 and Table 4.6 as well as the auxiliary notation in Table 4.4 it immediately follows that p provides an anticipation-based structure. Moreover, the recursively descending application of $code_{in}$ and $code_{out}$ ensures that p supports all expectations of $stmt'$. It is

$$c_{init}(s') = (h_\perp, \mathsf{v}, (\mathsf{v}_\perp, stmt')),$$

where v maps each global variable of s' to its initial value. Well-typedness of s' implies that

$$\Gamma_g; \Delta \vdash_{\mathsf{var}} (h_\perp, \mathsf{v}, (\mathsf{v}_\perp, stmt')),$$

where Γ_g represents the local type mapping regarding the global variables (cf. Rule T-Spec in Table 3.2). According to Definition C.3.7, it remains to show that the call stacks of the initial configurations of s and p are in relation regarding $\sim_{CS}$.

Case $stmt'$ is an active statement
In this case, consider

$$c_{init}(p) = (h_\perp, \mathsf{v}, (\mathsf{v}_\perp, stmt_{pl};\ \texttt{return}));$$

Since $stmt_{pl}$ results from applying $code_{out}$ to $stmt'$ we know from Lemma C.3.2 that

$$stmt' \sim_{st} stmt_{pl} \quad \text{hence} \quad (\mathsf{v}_\perp, stmt') \sim_{CS} (\mathsf{v}_\perp, stmt_{pl}).$$

Case $stmt'$ is a passive statement
In this case, consider

$$\overline{c_{init}}(p) = (h_\perp, \mathsf{v}, (\mathsf{v}_\perp, \epsilon));$$

Since $stmt'$ is an instance of s^{psv}, Definition C.3.3 yields

$$(\mathsf{v}_\perp, stmt') \sim_{CS} (\mathsf{v}_\perp, \epsilon).$$

□

Summary

In most of today's software development projects, testing is still the only feasible instrument for assuring the quality of a software product. Rigorous testing is necessary during all stages of the software development life cycle. While system and acceptance tests deal with the complete system, unit testing aims at the smallest building blocks of the software. Not only due to the growing popularity of agile software development methodologies, the responsibility for unit tests is more and more shifted from software testers to software developers. Besides the entailed advantages, like increasing quality awareness and immediate feedback to the programmers, this development also demands for novel testing frameworks accounting for the different qualifications and expectations of software developers. At the same time, the rampant use of object-oriented programming languages equally necessitates a change regarding the unit testing approaches. Specifically, instead of traditional *state-based* tests in terms of an input-output comparison, in an object-oriented context it is more useful to execute *interaction-* or *behavior-based* tests considering the *sequence of interactions* between the unit under test and its environment.

In this thesis we provide a unit testing approach for multi-purposes object-oriented programming languages in the style of *Java* and C♯. To meet the above indicated requirements our approach includes the definition of a test specification language which results from extending the programming language with new designated specification constructs. In this way, the software developer does not need to learn a completely new language. At the same time, adding new constructs allows to increase the abstraction of the language regarding the specification of interaction-based tests. In order to execute a specified test, programming language code is automatically generated from a test specification.

Our testing approach is presented in terms of a formal framework. On the one hand, this enables us to identify and analyze the requirements on the design of the specification language in a formal way. On the other hand, on the formal basis we can, likewise, give a formal definition of the code generation algorithm which, in turn, allows us to formally prove its correctness.

The development of the testing framework goes through several stages: First, we introduce a programming language that captures a reasonable subset of features many modern object-oriented general-purpose programming languages like *Java* and C♯ have in common. In particular, the language comes with a formal

operational semantics giving a precise meaning to programs without ambiguities.

After that, we discuss and justify the new specification constructs which we use in order to define the test specification language. A crucial aspect is that the specification language is also equipped with an operational semantics giving a precise meaning to the test specifications, as well.

Next, we present the code generation algorithm. To cope with the complexity, the code generation is divided into two parts. First, a preprocessing step enriches the original specification with additional code such that the new specification has some useful features. Second, the actual code generation step transforms the new test specification into a test program of the programming language. Finally, we provide a correctness proof regarding the code generation algorithm. In particular, we show that the resulting test program indeed tests for the specified interactions.

As mentioned above, the programming language represents only a small subset of languages like *Java* or C$^\sharp$. Also the corresponding test specification language lacks of several construct that would facilitate the writing of specifications. On account of this, we discuss some possible language extensions regarding, both, the programming language and the specification language.

An important feature of modern programming language is their support for *concurrency*. Therefore, we propose the introduction of concurrency into the programming language by means of *thread classes* which, in particular, allow to create new threads, dynamically. Finally we suggest a corresponding extension of the specification language and sketch a modification of the code generation algorithm.

Samenvatting

In de meeste software-ontwikkelingsprojecten van vandaag is testen nog steeds de enige bruikbare manier om de kwaliteit van een softwareproject te waarborgen. Rigoureus testen is nodig tijdens alle stages van de levenscyclus van de software-ontwikkeling. Terwijl systeem- en acceptatie-testen over het complete systeem gaan, is unit-testen gericht op de kleinste bouwstenen van de software. De verantwoordelijkheid voor de unit-testen is meer en meer verschoven van de softwaretesters naar de softwareontwikkelaars. Dit komt niet alleen door de groeiende populariteit van agile-software-ontwikkelingsmethodologieën. Naast de daaruit volgende voordelen voor de programmeurs, zoals het verhogen van hun kwaliteitsbewustzijn en onmiddellijke feedback, eist deze ontwikkeling ook nieuwe testraamwerken die rekening houden met de veranderde kwalificaties en verwachtingen van de softwareontwikkelaars. Tevens is door het sterk stijgende gebruik van object-georiënteerde programmeertalen ook een wijziging nodig wat betreft de unit-test benaderingen. In het bijzonder, in plaats van de traditionele toestand-gebaseerde tests door middel van een input-output vergelijking, is het in een object-georiënteerde context nuttiger om op interactie of gedrag gebaseerde tests uit te voeren, die op de volgorde van de interacties tussen de unit onder test en diens omgeving letten.

In dit proefschrift geven we een aanpak van unit-testen voor multi-purpose object-georiënteerde programmeertalen in de stijl van *Java* en C$^\sharp$. Om aan de bovengenoemde eisen te voldoen, bevat onze aanpak de definitie van een taal voor het specificeren van de tests. Die is het resultaat van de uitbreiding van de eerdergenoemde programmeertaal met nieuwe taalconstructies voor de specificatie van de tests. Op deze manier hoeft de softwareontwikkelaar niet een compleet nieuwe taal te leren. Tegelijkertijd maakt het toevoegen van nieuwe constructies het mogelijk om de abstractie van de taal te verhogen ten opzichte van de specificatie van interactie-gebaseerde tests. Om een bepaalde test uit te voeren wordt de programma-code automatisch van een test-specificatie gegenereerd.

Onze testaanpak wordt gepresenteerd op basis van een formeel raamwerk. Dit stelt ons enerzijds in staat om de eisen aan het ontwerp van de specificatietaal op een formele manier te identificeren en te analyseren. Anderzijds kunnen we op deze formele basis eveneens een formele definitie geven van het code-generatiealgoritme, die ons zijnerzijds in staat stelt om de correctheid ervan formeel te bewijzen.

De ontwikkeling van het testraamwerk gaat door verschillende fasen: ten eerste

introduceren wij een programmeertaal, die een redelijke deelverzameling van features omvat die veel moderne general-purpose object-georiënteerde programmeertalen, zoals *Java* en C♯, gemeen hebben. In het bijzonder komt de taal met een formele operationele semantiek, die een precieze betekenis aan programma's geeft, zonder dubbelzinnigheden.

Daarna bespreken en rechtvaardigen wij de nieuwe specificatie constructies die wij gebruiken om de test-specificatietaal te definiëren. Een belangrijk punt is dat de specificatietaal eveneens uitgerust is met een operationele semantiek die een precieze betekenis geeft aan de test specificaties.

Vervolgens presenteren wij het code-generatiealgoritme. Om de complexiteit het hoofd te bieden is de code-generatie gesplitst in twee delen. Het eerste gedeelte is een voorverwerkingsstap die extra code aan de oorspronkelijke specificatie toevoegt, zodat de nieuwe specificatie een aantal handige features heeft. De tweede stap is de werkelijke code-generatie, die de nieuwe test-specificatie transformeert naar een testprogramma van de programmeertaal. Tot slot presenteren we een correctheidsbewijs van het code-generatiealgoritme. In het bijzonder tonen we aan dat het resulterende testprogramma inderdaad voor de opgegeven interacties test.

Zoals hierboven vermeld, vertegenwoordigt de programmeertaal slechts een kleine deelverzameling van talen zoals *Java* of C♯. Eveneens ontbreken een aantal constructies in de test-specificatietaal die het schrijven van bepaalde specificaties makkelijker zouden maken. Op grond daarvan bespreken wij een aantal mogelijke taaluitbreidingen voor zowel de programmeertaal als de specificatietaal.

Een belangrijke feature van de moderne programmeertalen is hun ondersteuning voor concurrency. Daarom stellen we de invoering voor van concurrency in de programmeertaal door middel van threadklassen die, in het bijzonder, het mogelijk maken om nieuwe threads dynamisch te creëren. Tot slot stellen we daarbij een uitbreiding voor van de specificatietaal en schetsen we een wijziging van het code-generatiealgoritme.

Curriculum Vitæ

Personal Data

Name	Andreas Grüner (birth name: Lukosch)
Civil status	Married, one child
Nationality	German
Date of Birth	03.04.1974
Place of Birth	Nettetal

Carrier

10/2009–	Head of the *@rtus* software development group at *Dataport*
04/2008–03/2009	Doctorand at the University of Leiden, member of the research cluster *Foundations of Software Technology*
04/2005–04/2008	Doctorand at the Chair of Software Technology, University of Kiel, as member of the NWO/DFG-project *Mobi-J: Formal Methods for Component and Objects*
12/1999–12/2003	Software engineer at Bartsch-Software
11/1999–04/2000	Student assistant to the Chair of Multimedia Information Processing at the University of Kiel

Education

10/1999–12/2004	University of Kiel, graduate studies in computer science (Diplom), minor subject in mathematics. Diplom thesis: *Cliques and Components: Implementing Traces and Object Connectivity for a Concurrent Language*, Supervisors: Prof. Dr. W.-P. de Roever, Dr. M. Steffen
10/1994–06/1996	University of Kiel, studies in physics
06/1984–06/1994	Hans-Geiger-Gymnasium Kiel, highschool diploma (Abitur) Primary school in Kiel and Nettetal

Index

accessibility, 59
activation record
  CoJapl, 141
  Japl, 45
    specification language, 73
  sequential programming language, 31
active statement, *see* control context, active
anticipation, 87, 92, 159, 194
anticipation-based code structure, 205
anticipation-valid
  code, 194
  configuration, 197
anticipation-validity, 197
auxiliary notations, 34

balance, 43
behavior-based testing, *see* interaction-based testing
binary tree, 26
bisimulation, 106
  testing, 109, 208
  weak, 107, 189

C♯, 4
C^{++}, 3
call stack
  CoJapl, 141
    specification language, 157
  Japl, 46
    specification language, 73
  sequential programming language, 31
code generation, 99
code generation, 83, 159, 187
  Japl, 95, 102
    correctness, 103
  code-in, 101
  code-out, 100
  correctness, 203, 217
CoJapl, 139
communication label, *see* transition label
completeness, 58
component, 38
compositionality, 54, 177
concurrency, 139
configuration
  CoJapl, 143
  Japl, 45
    specification language, 73
  initial, 36, 76, 149
    active, 50
    passive, 50
  sequential programming language, 31
consistent control flow, 43
consistent information flow, 43
control context
  active, 67
  passive, 67
control flow, 84

decomposition, 184
defect, 6

environment, 38
error, 6
executability, 57, 79

expectation body, 63
expectation statement, 59
expression evaluation, 34
external semantics
  Japl, 41

fail, *see* failure
failure, 6
failure report, 111
faulty specification, 111
free variables, 46

incoming communication, 41
inheritance, 126
initial expectation identifier, adjustment, 92
input enabledness, 79
integration testing, 5
interaction-based testing, 11, 57
interactions, 10
interface communication, 41
internal semantics, 41

Java, 3
Japl, 38
JMLUnit, 14
jMock, 10
JUnit, 7

label check, 45
labeled transition system, 41
labeling mechanism, 85

merge
  components, 54
  configurations, 179
mock class specification, 65
mock objects, 11, 65
mock thread specification, 153
mocks, *see* mock objects
mutual exclusion, 161

next, 87
*next*list, 160

object-oriented programming, 2
operational semantics, 36
  CoJapl, 145–147, 150
    specification language, 159
  Japl, 48
    specification language, 73
  sequential programming language, 31
  specification classes, 119
  subtyping and inheritance, 131, 133
outgoing communication, 41

passive statement, *see* control context, passive
preprocessing, 85, 89, 90, 95, 187
program execution
  CoJapl, 149
  Japl, 50
  sequential programming language, 36
programming classes, 122
propagation of new names, 49

realizability
  Japl, 42
renaming, 49

satisfiability, 58
Simula, 2
simulation, 104
  testing, 108
  weak, 105
Smalltalk, 3
software crisis, 1
software development process, *see* software development life-cycle
spawn, 139, 152
specification language, *see* test specification language
specification class, 115
specification execution, 75
state-based testing, 52
static semantics, 29, 141
  CoJapl, 141
    specification language, 157
  Japl, 40

specification language, 68
programming classes, 124
sequential programming language, 26
specification classes, 118
subtyping and inheritance, 128
subclassing, 126
subject reduction, 50, 173
subtyping, 126
syntax
CoJapl, 140
specification language, 155
Japl, 39
specification language, 64
programming classes, 122
sequential programming language, 24
specification classes, 116
subtyping and inheritance, 127

test specification language, 57, 115, 122, 151
test thread specification, 152, 153
thread class, 139
thread configuration mapping, 143
thread-save, 156
trace
Japl, 50
specification language, 75
trace semantics
CoJapl, 149
Japl, 51
specification language, 76
traces, 50
transition label
CoJapl, 146
Japl, 42
type system, *see* static semantics
types
sequential programming language, 29

unit testing, 5, 7

variable binding, 84, 93
variable evaluation, 33
variable globalization, 94
V-model, 5

well-typedness, dynamic specification code, 207