Testing object Interactions

Grüner, A.

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

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

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

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

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:

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

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:

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

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

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

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

s	$::= \overline{cutdecl}\ \overline{T}\ \overline{x};\ \overline{mokdecl}\ \{\ stmt\ \}$	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
$stmt$	$::= x = e \mid x = \texttt{new}\ C() \mid \varepsilon \mid stmt; stmt \mid \{\overline{T}\ \overline{x};\ stmt\}$	statements
	$\mid \texttt{while}\ (e)\ \{stmt\} \mid \texttt{if}\ (e)\ \{stmt\}\ \texttt{else}\ \{stmt\}$	
	$\mid stmt_{in} \mid stmt_{out} \mid \texttt{case}\ \{\ \overline{stmt_{in}; stmt}\ \}$	
$stmt_{in}$	$::= (C\ x)?m(\overline{T}\ \overline{x}).\texttt{where}(e)\ \{\overline{T}\ \overline{x};\ stmt;\ !\texttt{return}\ e\}$	incoming stmt
	$\mid \texttt{new}(C\ x)?C(\overline{T}\ \overline{x}).\texttt{where}(e)\ \{\overline{T}\ \overline{x};\ stmt;\ !\texttt{return}\}$	
$stmt_{out}$	$::= e!m(e,\ldots,e)\ \{\overline{T}\ \overline{x};\ stmt;\ x = ?\texttt{return}(T\ x).\texttt{where}(e)\ \}$	outgoing stmt
	$\mid \texttt{new}!C(e,\ldots,e)\ \{\overline{T}\ \overline{x};\ stmt;\ x = ?\texttt{return}(T\ x).\texttt{where}(e)\ \}$	
e	$::= x \mid \texttt{null} \mid \texttt{op}(e,\ldots,e)$	expressions

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

$$s_{out} = o_1!m_1(\overline{v})\{\ \overline{T_l}\ \overline{x_l};\ stmt_1;\ x = ?\texttt{return}(T\ x').\texttt{where}(e)\ \},$$

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

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

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

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

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

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

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

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

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

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

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

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

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:

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

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.

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

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 $\mathsf{v}_l = \{\overline{x} \mapsto ival(\overline{T})\}$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

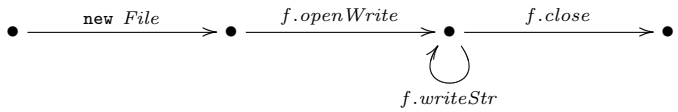
3.5 Example

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

```
1  File file;
2  bool writing;
3
4  mock class File{ File(string);
5                   bool openWrite();
6                   string writeStr(string);
7                   bool close()
8                 }
9
10 {
11   new(File f)?File(string fname).where(fname != "") {
12     file = f;
13     !return;
14   };
15   file?openWrite() {
16     writing = true;
17     !return(writing);
18   };
19   while (writing) {
20     case {
21       file?writeStr(string s) {
22         !return(s);
23       }
24       file?close() {
25         writing = false;
26         !return(true)
27       }
28     }
29   }
30 }
```

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 $\bar{t} \in [\![\Theta \vdash s : \Delta]\!]$.