Testing object Interactions

Grüner, A.

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

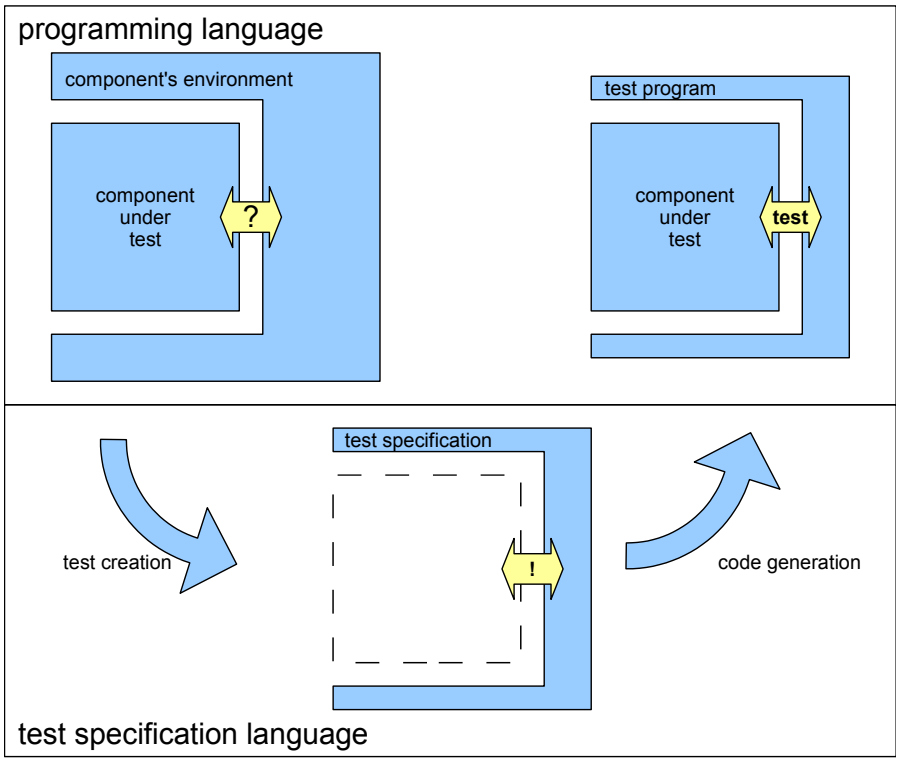

Figure 4.1: Testing framework

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

$$prep_{in} : s^{psv} \times s_{nxt} \to s_{nxt} \times s^{psv} \quad \text{and}$$
$$prep_{out} : s^{act} \to s^{act},$$

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.

$$prep_{out}(e!m(e,\ldots,e)\{\overline{T}\,\overline{x};\ s_1^{psv};\ x = ?\texttt{return}(T\,x').\texttt{where}(e')\ \}) \stackrel{\text{def}}{=}$$
$$\quad s_{nxt};\ e!m(e,\ldots,e)\{\overline{T}\,\overline{x};\ s_2^{psv};\ x = [i]?\texttt{return}(T\,x').\texttt{where}(e')\};\ check(i,e');$$
$$\qquad \text{where}\ \ (s_{nxt}, s_2^{psv}) = prep_{in}(s_1^{psv}, \texttt{next} = i)$$

$$prep_{out}(\texttt{new!}C(e,\ldots,e)\{\overline{T}\,\overline{x};\ s_1^{psv};\ x = ?\texttt{return}(C\,x').\texttt{where}(e')\ \}) \stackrel{\text{def}}{=}$$
$$\quad s_{nxt};\ \texttt{new!}C(e,\ldots,e)\{\overline{T}\,\overline{x};\ s_2^{psv};\ x = [i]?\texttt{return}(C\,x').\texttt{where}(e')\};\ check(i,e');$$
$$\qquad \text{where}\ \ (s_{nxt}, s_2^{psv}) = prep_{in}(s_1^{psv}, \texttt{next} = i)$$

$$prep_{out}(\texttt{if}\ (e)\ \{s_1^{act}\}\ \texttt{else}\ \{s_2^{act}\}) \stackrel{\text{def}}{=}$$
$$\quad \texttt{if}\ (e)\ \{prep_{out}(s_1^{act})\}\ \texttt{else}\ \{prep_{out}(s_2^{act})\}$$

$$prep_{out}(\texttt{while}\ (e)\ \{s^{act}\}) \stackrel{\text{def}}{=} \texttt{while}\ (e)\ \{prep_{out}(s^{act})\}$$

$$prep_{out}(s_1^{act};\ s_2^{act}) \stackrel{\text{def}}{=} prep_{out}(s_1^{act});\ prep_{out}(s_2^{act})$$

$$prep_{out}(\{\overline{T}\,\overline{x};\ s^{act}\}) \stackrel{\text{def}}{=} \{\overline{T}\,\overline{x};\ prep_{out}(s^{act})\}$$

$$prep_{out}(x = e) \stackrel{\text{def}}{=} x = e$$

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

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

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

$$prep_{in}(\texttt{case}\ \{\overline{stmt_{in}; s^{psv}}\}, s_{nxt}) \stackrel{\text{def}}{=} (\ next = i,\ \texttt{case}\ \{\overline{\tilde{stmt}_{in}; \tilde{s}^p}\}\ )$$
where for each $stmt_{in}^l; s_l^{psv} \in \overline{stmt_{in}; s^{psv}}$
$stmt_{in}^l = (C\ x)?m(\overline{T}\ \overline{x}).\texttt{where}(e)\{\overline{T}\ \overline{x}; s^{act};\ \texttt{!return}\ e'\}$ it is
$(s_{nxt}^l, \tilde{s}_l^p) = prep_{in}(s_l^{psv}, s_{nxt})$ 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 $\{\ \overline{T}\,\overline{x};\ stmt\ \}$ 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

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

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

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

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$_1$
⋮
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, $x =$ `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:
$T\ meth(T_1\ x_1, \ldots, T_k\ x_k)$ {	$C(T_1\ x_1, \ldots, T_k\ x_k)$ {
$T\ retVal$;	
	`if`($internal$ == `true`) {
	$skip$;
$fail$;	} `else` { $fail$; }
`return`($retVal$);	`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

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

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

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

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.

$$code_{out}(\ \overline{cldef},\ e!m(\overline{e})\ \{stmt;\ [i]?\texttt{return}(x).\texttt{where}(e_w)\}\ ) \stackrel{\text{def}}{=}$$
$$\quad \textsf{let}\ \ \overline{cldef'} = code_{in}(\overline{cldef}, stmt)\ \ \textsf{in}\ (\ \overline{cldef'},\ x = e.m(\overline{e})\ ).$$

$$code_{out}(\ \overline{cldef},\ \texttt{new}!C(\overline{e}\ )\{stmt;\ [i]?\texttt{return}(x).\texttt{where}(e_w)\}) \stackrel{\text{def}}{=}$$
$$\quad \textsf{let}\ \ \overline{cldef'} = code_{in}(\overline{cldef}, stmt)\ \ \textsf{in}\ (\ \overline{cldef'},\ x = \texttt{new}\ C(\overline{e})\ ).$$

$$code_{out}(\ \overline{cldef},\ x = e\ ) \stackrel{\text{def}}{=} (\ \overline{cldef},\ x = e\ )$$

$$code_{out}(\ \overline{cldef},\ \texttt{new}\ C()\ ) \stackrel{\text{def}}{=}$$
$$\quad \textsf{let}\ \ \overline{T}\,\overline{x} = cparams(C)\ \ \textsf{in}$$
$$\quad\quad \textsf{let}\ \ stmt = intern = \texttt{true};\ x = \texttt{new}\ C(ival(\overline{T}));\ intern = \texttt{false}\ \ \textsf{in}\ (\ \overline{cldef},\ stmt\ ).$$

$$code_{out}(\ \overline{cldef},\ stmt_1;\ stmt_2\ ) \stackrel{\text{def}}{=}$$
$$\quad \textsf{let}\ \ (\overline{cldef}_1, stmt_1^{pl}) = code_{out}(\overline{cldef}, stmt_1)\ \ \textsf{in}$$
$$\quad\quad \textsf{let}\ \ (\overline{cldef}_2, stmt_2^{pl}) = code_{out}(\overline{cldef}_1, stmt_2)\ \ \textsf{in}\ (\ \overline{cldef}_2,\ stmt_1^{pl};\ stmt_2^{pl}\ ).$$

$$code_{out}(\ \overline{cldef},\ \texttt{while}\ (e)\ \{stmt\}\ ) \stackrel{\text{def}}{=}$$
$$\quad \textsf{let}\ \ (\overline{cldef'}, stmt^{pl}) = code_{out}(\overline{cldef}, stmt)\ \ \textsf{in}\ (\ \overline{cldef'},\ \texttt{while}\ (e)\ \{stmt^{pl}\}\ ).$$

$$code_{out}(\ \overline{cldef},\ \texttt{if}\ (e)\ \{stmt_1\}\,\texttt{else}\ \{stmt_2\}\ ) \stackrel{\text{def}}{=}$$
$$\quad \textsf{let}\ \ (\overline{cldef}_1, stmt_1^{pl}) = code_{out}(\overline{cldef}, stmt_1)\ \ \textsf{in}$$
$$\quad\quad \textsf{let}\ \ (\overline{cldef}_2, stmt_2^{pl}) = code_{out}(\overline{cldef}_1, stmt_2)\ \ \textsf{in}\ (\overline{cldef}_2,\ \texttt{if}(e)\ \{stmt_1^{pl}\}\,\texttt{else}\{stmt_2^{pl}\})$$

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}
code_{in}(\overline{cldef}, [i]\,(C\;x)?m(\overline{T}\;\overline{x}).\texttt{where}(e)\{check(i,e);\;\; stmt;\;\; !\texttt{return}\;e_r\}) \stackrel{\text{def}}{=} \\
\quad \mathsf{let}\;\; (\overline{cldef'}, stmt^{pl}) = code_{out}(\overline{cldef}, stmt)\;\; \mathsf{in} \\
\qquad \mathsf{let}\;\; \overline{T}\;\overline{x}_p = mparams(C,m)\;\; \mathsf{in} \\
\qquad\quad \mathsf{let}\;\; (e', e'_r, stmt^{pl\prime}) = (e, e_r, stmt^{pl})[\texttt{this}/x,\;\; \overline{x}_p/\overline{x}]\;\; \mathsf{in}\;\; \overline{cldef'}.C.m \stackrel{(i,e)}{\rightarrow} stmt^{pl\prime} : e'_r. \\
\\
code_{in}(\overline{cldef}, [i]\,\texttt{new}(C\;x)?C(\overline{T}\;\overline{x}).\texttt{where}(e)\{check(i,e);\;\; stmt;\;\; !\texttt{return}\}) \stackrel{\text{def}}{=} \\
\quad \mathsf{let}\;\; (\overline{cldef'}, stmt^{pl}) = code_{out}(\overline{cldef}, stmt)\;\; \mathsf{in} \\
\qquad \mathsf{let}\;\; \overline{T}\;\overline{x}_p = cparams(C)\;\; \mathsf{in} \\
\qquad\quad \mathsf{let}\;\; (e', stmt^{pl\prime}) = (e, stmt^{pl})[\texttt{this}/x,\;\; \overline{x}_p/\overline{x}]\;\; \mathsf{in}\;\; \overline{cldef'}.C.C \stackrel{(i,e)}{\rightarrow} stmt^{pl\prime}. \\
\\
code_{in}(\overline{cldef}, stmt_1;\;\; stmt_2) \stackrel{\text{def}}{=} \\
\quad \mathsf{let}\;\; \overline{cldef}_1 = code_{in}(\overline{cldef}, stmt_1)\;\; \mathsf{in} \\
\qquad \mathsf{let}\;\; \overline{cldef}_2 = code_{in}(\overline{cldef}_1, stmt_2)\;\; \mathsf{in}\;\; \overline{cldef}_2. \\
\\
code_{in}(\overline{cldef}, \texttt{if}(e)\;\{stmt_1\}\;\texttt{else}\;\{stmt_2\}) \stackrel{\text{def}}{=} \\
\quad \mathsf{let}\;\; \overline{cldef}_1 = code_{in}(\overline{cldef}, stmt_1)\;\; \mathsf{in} \\
\qquad \mathsf{let}\;\; \overline{cldef}_2 = code_{in}(\overline{cldef}_1, stmt_2)\;\; \mathsf{in}\;\; \overline{cldef}_2. \\
\\
code_{in}(\overline{cldef}, \texttt{while}(e)\;\{stmt\}) \stackrel{\text{def}}{=} \\
\quad \mathsf{let}\;\; \overline{cldef}_1 = code_{in}(\overline{cldef}, stmt)\;\; \mathsf{in}\; \overline{cldef}_1. \\
\\
code_{in}(\overline{cldef}, \texttt{case}\;\{\;stmt_1\;\; stmt_2\;\; \ldots\;\; stmt_n\}) \stackrel{\text{def}}{=} \\
\quad \mathsf{let}\;\; \overline{cldef}_1 = code_{in}(\overline{cldef}, stmt_1)\;\; \mathsf{in} \\
\qquad \ddots \\
\qquad \mathsf{let}\;\; \overline{cldef}_n = code_{in}(\overline{cldef}_{n-1}, stmt_n)\;\; \mathsf{in}\;\; \overline{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

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

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

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

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 ($\stackrel{a}{\Longrightarrow}$) 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 $S \in \mathit{Conf} \times \mathit{Conf}$ is a *bisimulation* if for all pairs of configurations $c, d \in \mathit{Conf}$ the following holds:

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

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

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

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

   (a) If
   $$d \rightsquigarrow d'$$
   then there is a $c' \in \mathit{Conf}$ 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 $c' \in \mathit{Conf}$ 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 $c' \in \mathit{Conf}$ with

$$c \rightsquigarrow c'$$

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

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

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

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

implies that

(a) either there is a $d' \in \mathit{Conf}$ 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 $d' \in \mathit{Conf}$ 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 $S \in \mathit{Conf} \times \mathit{Conf}$ 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.