Testing object Interactions

Grüner, A.

Chapter 5

Further possible extensions

Within the last chapters we have provided a basic framework for testing components of object-oriented class-based languages like *Java* and C♯. A main contribution was the development of a test specification language which allows to specify a desired interface trace in order to stipulate the expected interface behavior of an object-oriented component under test. As mentioned earlier, however, some of the common features of object-oriented programming languages have been omitted. In this chapter we want to discuss some of these features. In particular, we sketch possible approaches to incorporating certain features into our programming language. We also investigate the extension's impact on our testing approach and correspondingly suggest additional modifications of the specification language, if necessary. Furthermore, we discuss some extensions concerning the specification language only, that is, language features that may facilitate writing test specification.

5.1 Specification classes

We have introduced a test specification language which can be used to describe expected interface interactions of communicating *objects*. The specification language itself, however, is not object-oriented. Extending the specification language with classes and objects may allow for reusing and parameterizing specifications.

Specifically, in this section we want to investigate an extension of the specification language with *specification classes*. Method bodies of specification classes consist of specification statements. An invocation of such a specification method gives rise to the expectation of the interface interaction sequence given by the method's body. A specification statement within a method body might contain reference to fields and to parameters of the method as well as calls to other specification methods. In particular, a specification method may call itself, i.e., the extension of the language introduces recursion. Summarizing, a method body of

s	$::= \overline{cutdecl}\ \overline{T}\ \overline{x};\ \overline{mokdecl}\ \overline{cldef}\ \{\ stmt\ \}$	specification
$cutdecl$	$::= \texttt{test class}\ C;$	test unit class
$mokdecl$	$::= \texttt{mock class}\ C\{C(T,\ldots,T);\ \overline{T\ m(T,\ldots,T)}\};$	mock class
$cldef$	$::= \texttt{class}\ C\{\overline{T}\ \overline{f};\ con\ \overline{mdef}\}$	class def.
con	$::= C(\overline{T}\ \overline{x})\{\overline{T}\ \overline{x};\ stmt;\ \texttt{return}\}$	constructor
$mdef$	$::= X\ m(\overline{T}\ \overline{x})\{\overline{T}\ \overline{x};\ stmt;\ \texttt{return}\}$	meth. def.
$stmt$	$::= x = e \mid x = \texttt{new}\ C() \mid \varepsilon \mid stmt; stmt \mid \{\overline{T}\ \overline{x};\ stmt\}$	statements
	$\mid \texttt{while}\ (e)\ \{stmt\} \mid \texttt{if}\ (e)\ \{stmt\}\ \texttt{else}\ \{stmt\}$	
	$\mid stmt_{in} \mid stmt_{out} \mid \texttt{case}\ \{\ \overline{stmt_{in}; stmt}\ \}$	
	$\mid f = e \mid e.m(e,\ldots,e) \mid x = \texttt{new}\ C(e,\ldots,e)$	
$stmt_{in}$	$::= (C\ x)?m(\overline{T}\ \overline{x}).\texttt{where}(e)\ \{\overline{T}\ \overline{x};\ stmt;\ !\texttt{return}\ e\}$	incoming stmt
	$\mid \texttt{new}(C\ x)?C(\overline{T}\ \overline{x}).\texttt{where}(e)\ \{\overline{T}\ \overline{x};\ stmt;\ !\texttt{return}\}$	
$stmt_{out}$	$::= e!m(e,\ldots,e)\ \{\overline{T}\ \overline{x};\ stmt;\ ?\texttt{return}(x).\texttt{where}(e)\}$	outgoing stmt
	$\mid \texttt{new}!C(e,\ldots,e)\ \{\overline{T}\ \overline{x};\ stmt;\ ?\texttt{return}(x).\texttt{where}(e)\}$	
e	$::= x \mid f \mid \texttt{this} \mid \texttt{null} \mid \texttt{op}(e,\ldots,e)$	expressions
X	$::= ? \mid !$	control context

Table 5.1: Extension by specification classes: syntax

a specification class represents a trace specification which is possibly abstracted over parameters, variables, and sub-traces.

Note that introducing specification classes, renders it necessary to distinguish them from mock classes. While mock classes are still given in terms of their signature only, specification classes in contrast are fully-fletched classes similar to the programming language classes except that their method bodies may contain expectation statements. Furthermore, mock objects still neither provide fields nor do they allow for internal method calls. On the contrary, specification classes and their objects must not show up at the interface trace, hence, they can be considered as hidden classes with respect to the specification program's environment. We will capture these requirements by the type system, which we will explain somewhat later.

Table 5.1 suggests a grammar extension of the specification language regarding specification classes. We have extended the grammar of the original specification language given in Table 3.1 by constructs for class definitions and by statements for method and constructor calls as well as field updates. The definition of specification classes resembles the definition of conventional classes in the programming language but differs in two aspects. First, for simplicity reasons, method definitions do not include return values and therefore not a return type either. Second, instead of a return type, method definitions state the control context X of the body statement. Note, that a method call statement does not entail an assignment, due to the lack of a return value. Finally, we have added rules for expressions that yield the current object's name or the value of one of its fields, respectively.

We said earlier, that we have to distinguish mock and specification classes. The proper differentiation will be carried out by the type system. In particular, internal method calls as well as field accesses may only be targeted at instances of specification classes, while interface communication statements may only involve external or mock classes.

Additionally, we have to ensure that the new constructs do not allow to specify an inconsistent control flow. For, in general we want to allow invocations of specification methods to appear within, both, passive and active control context, yet we have to check that a specification method's body complies with the control context of its call. In account of this, we extend the type definition given in Definition 2.2.1 by adding the rule

$$T ::= (\mathit{MNames} \cup \mathit{CNames}) \rightharpoonup (U \times \ldots \times U)^{\gamma}.$$

The new type is used for specification classes. It yields the parameter types and the control context γ of each of the specification class's methods and constructor, allowing to check for control flow consistency. At the same time, it also allows to distinguish mock and specification classes, as mock classes will be associated with the usual class types.

The type system of the original specification language is extended by rules for the new constructs. These new rules resemble the corresponding typing rules of the programming language as given in Table 2.2 and Table 2.3. Besides adding new rules we only have to modify the Rule T-Spec in order to deal with the new class definitions. Thus, Table 5.2 only shows the new version of Rule T-Spec as well as the new rules concerning the new class definition constructs and the new statements. All other rules that were given in Table 3.2 are inherited without any modifications and are, therefore, left out. We also omit the straightforward typing rules for the new expressions.

As mentioned earlier, we extend Rule T-Spec with a judgment for type checking the specification class definitions. Additionally, we have to ensure that specification types do not show up in the interface communication. That is, the signatures of methods and constructors of both, mock classes and imported classes, must not include class names of specification classes. This check is abbreviated by the new premise $\Delta \vdash \Theta : \mathsf{ok}$, which stands for:

$$\begin{aligned}
&\forall C \in \mathit{dom}(\Delta).\ \forall C' \in \mathsf{commedCl}(C, \Delta).\ C' \in \mathit{dom}(\Theta) \Rightarrow \mathsf{isMockCl}(C') \\
&\wedge \\
&\forall C \in \mathit{dom}(\Theta).\ \mathsf{isMockCl}(C) \Rightarrow \forall C' \in \mathsf{commedCl}(C, \Theta). \\
&\quad C' \in \mathit{dom}(\Theta) \Rightarrow \mathsf{isMockCl}(C'),
\end{aligned}$$

where we use the following two auxiliary functions in order to determine the set of communicated class names within the signature of a given class and to find out if a class is a mock class but not a specification class:

$$\mathsf{commedCl}(C, \chi) \stackrel{\text{def}}{=} \cup_{m \in \mathit{dom}(C)} \{\chi(C)(m).\mathit{dom} \cup \chi(C)(m).\mathit{ran}\} \quad \text{and}$$

$$\mathsf{isMockCl}(C) \stackrel{\text{def}}{=} \Theta(C) \in (\mathit{FNames} \cup \mathit{CNames}) \rightharpoonup (T \times \ldots \times T \to T)$$

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

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

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

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

$$[\text{T-Call}_{\text{SCl}}]\ \frac{\Gamma;\Delta,\Theta \vdash e : C \quad \Theta(C)(m) = \overline{T}^{\gamma} \quad \Gamma;\Delta \vdash \overline{e} : \overline{T}}{\Gamma;\Delta;\Theta \vdash e.m(\overline{e}) : \mathsf{ok}^{\gamma}}$$

$$[\text{T-New}_{\text{SCl}}]\ \frac{\Theta(C)(C) = \overline{T}^{act} \quad \Gamma;\Delta,\Theta \vdash \overline{e} : \overline{T}}{\Gamma;\Delta;\Theta \vdash x = \texttt{new}\ C(\overline{e}) : \mathsf{ok}^{act}} \qquad [\text{T-FUpd}_{\text{SCl}}]\ \frac{\Gamma;\Delta,\Theta \vdash e : \Gamma(f)}{\Gamma;\Delta;\Theta \vdash f = e : \mathsf{ok}^{act}}$$

Table 5.2: Extension by specification classes: type system (stmts)

In words, each externally defined class shall communicate only instances of tester classes that are indeed mock classes. Likewise, each mock class shall only communicate instances of tester classes that are mock classes, too.

The new typing Rule T-SClass deals with specification classes and is almost identical to Rule T-Class for programming classes except that the assumed typing context is conformed to the typing context of the specification language's type system. Note, that a class definition is well-typed in a passive and in an active control context. Also constructor and method definitions are well-typed in any control context as can be seen from Rule T-SCon and Rule T-SMDef, respectively. The body of a constructor definition, however, is only well-typed in an active control context; a method body is type-checked in the control context that has been stated in the method definition. Consequently, an internal method call is only well-typed if it occurs in a control context which corresponds to the control context of the called method (T-CallSCl). This check also ensures that no mock method can be called internally, as mock classes do not provide control contexts for their methods at all. An instantiation of a specification class is handled in Rule T-NewSCl. It may only occur in an active control context as it involves a side-effect in form of a variable update. For the same reason, also field updates are only allowed in an active control context, as can be seen in Rule T-FUpdSCl.

Concerning the operational semantics, we can leave the rules regarding the interface communication as given in Table 3.3 in Section 3.4. For, the specification classes must not make any contributions to the interface communication. As for the internal computation steps, the operational semantics has to be extended by new rules for the three new statements, namely for field updates, method calls and constructor calls of specification classes. Additionally, we have to add rules for the return from internal method and constructor calls. Fortunately, we can borrow the corresponding internal rules of the programming language of Table 2.7 with almost no modifications. We only have to simplify Rule CALL and RET, since in the specification language the internal calls do not return values. The new rules are given in Table 5.3. Finally, the new constructs do not entail new types of interface communications, hence, we do not have to extend or modify the transition rules dealing with the interface communication.

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

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

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

$$[\text{RET}_m]\ (h,\mathsf{v},(\mu,\texttt{return})\circ(\mu',\texttt{rcv};\ mc)\circ\mathsf{CS}^b) \rightsquigarrow (h,\mathsf{v},(\mu',mc)\circ\mathsf{CS}^b)$$

$$[\text{RET}_c]\ \frac{(\mathsf{v}',\mu'') = vupd(\mathsf{v},\mu', x \mapsto [\![e]\!]_h^{\mathsf{v},\mu})}{(h,\mathsf{v},(\mu,\texttt{return}\ e)\circ(\mu',\texttt{rcv}\ x;\ mc)\circ\mathsf{CS}^b) \rightsquigarrow (h,\mathsf{v}',(\mu'',mc)\circ\mathsf{CS}^b)}$$

Table 5.3: Extension by specification classes: operational semantics

Although extending the specification language by specification classes was more or less straightforward, the code generation algorithm gets considerably more complex. This has three reasons. First, introducing recursion entails the possibility that several instances of an expectation statement's local variable exist in the variable stack at the same time, rendering it impossible to replace them by global variables. The same applies to the parameters of an expectation statement. Consider for instance a specification that contains the following method definition:

```
?specMeth(C o1) {
  o1?mockMeth(C o2, D o3){
    o3!unitMeth(){
```

```
      this.specMeth(o2);
      ?return()
    };
    !return(o3)
  };
  return
}
```

The body of the specification method *specMeth* represents the expectation of an incoming call of *mockMeth*, which is not answered with an immediate return but provokes a call-back of method *unitMeth* to the component under test. The body of the call-back specification, in turn, contains a recursive call of the specification method *specMeth*.

This example makes clear, that introducing global variables for the incoming call's parameters *o2* and *o3* is not sufficient, as the execution of the specification may lead to two instances of the incoming call statement of *mockMeth* on the call stack, where each instance needs its own parameter representation in the variable stack. As a consequence, the parameters and local variables of interaction statements cannot be replaced by global variables anymore but one has to emulate the variable stack of the called methods.

The second complication concerns the update of the global variable *next* which we used to anticipate the next incoming communication. A specification program may contain several internal call statements referring to the same specification method, such that each call statement requires a different update of *next* after the specification method has been processed. For a better understanding, consider the following specification snippet:

```
?specMeth() {
  (C x)?mockMeth1(){ !return() };
  return
}
⋮
{ // main body of specification :
  specMeth(); (D y)?mockMeth2() { !return() };
  specMeth(); (E z)?mockMeth3() { !return() }
}
```

Thus, we have defined a specification methods *specMeth*, which is called by the main body twice, such that each call is followed by the incoming call statement of another mock method (namely *mockMeth2* and *mockMeth3*, respectively). Following the labeling approach suggested in Chapter 4, we would equip the incoming call of *mockMeth2* and *mockMeth3* with expectation ids, say, i_2 and i_3. Moreover, we would have to insert an update of *next* in front of the last outgoing communication term that precedes the incoming call statements. In our example, however, both the incoming call statements, *mockMeth2* and *mockMeth3*, are preceded by the outgoing return term in the specification method *specMeth*. Thus, we cannot determine the identifier of the next expected incoming call statically, as *specMeth*

is called more than once. A solution to this problem is to adapt the preprocessing steps such that we add a parameter to each specification method for incoming communications. The parameter is used to determine the desired update statement for *next*. Thus, regarding our example, the outcome of the preprocessing could be sketched in the following way:

```
?specMeth(int updatebranch) {
  [i1](C x)?mockMeth1(){
    if (updatebranch == 1) { nxt = i2 }
    else { };
    if (updatebranch == 2) { nxt = i3 }
    else { };
    !return };
  return
}
⋮
{ // main body of specification :
  ...
  specMeth(1); [i2](D y)?mockMeth2() { ... !return };
  specMeth(2); [i3](E z)?mockMeth3() { ... !return }
 }
```

The third complication arise from the fact that specification methods for passive control contexts, i.e., a specification method whose body starts with an incoming call statement, cannot be translated to a corresponding method in the programming language. Again consider a small example

```
?specMeth(C x) {
  x?mockMeth(){ ... !return };
  return
}
⋮
{ // main body of specification :
  ... o1!unitMeth() { specMeth(o2); ... ?return }
}
```

In this example, a specification method *specMeth* is given whose body consists of an incoming call statement where the expected callee is determined by the specification method's parameter x. An invocation of method *specMeth* happens in the main body right after an *outgoing* call term. Thus, this internal call cannot be carried out in the programming language as it does not allow internal computation steps right after an outgoing communication. Moreover, we know from Chapter 4 that the incoming call statement in *specMeth* will be translated to a fragment of the method definition of method *mockMeth*. However, certainly the method *mockMeth* won't have direct access to parameters of the specification method.

To overcome these problems, we suggest the following approach. According to Rule Call in Table 5.3, the invocation of a specification method results into a

new activation $(\mathsf{v}_l, mbody(C)(m)$ which may evolve to some (μ, mc), in general. In the translated code, we emulate the call of a specification method that appears within passive control contexts by providing a global variable *specMethVars* which captures the variable function lists μ of its activation records. Since passive specification methods do not contain active statement that have to be carried out by the specification method, we do not need a representation of the activation record's code *mc*. The variable *specMethVars* consists of a list of structures where each structure contains the a specification method's local variable list μ including its actual parameters as well as a reference to the corresponding specification object. Accesses to the parameters and local variables of a specification method are replaced by accesses to the structure. Accesses to fields of a specification object are replaced by calls to designated access methods.

Now, a call of a passive specification method has to be anticipated such that a corresponding structure is created right before the preceding outgoing communication happens. Correspondingly, prior to the last outgoing communication term within the specification method the structure of the specification method has to be deleted. Hence, we have to extend the anticipation mechanism of Section 4.1 such that is does not only handle the anticipation of incoming call expectations but also the anticipation regarding invocations of specification methods that entail an incoming call expectation.

5.2 Programming classes

The previous section has shown that extending the specification language with specification classes requires a complex adaption of the code generation process. Extending the specification language with programming language classes, in contrast, only involves a rather moderate adaption. More specifically, we want to allow the usage of classes whose method bodies do not contain any specification statements but only programming language statements. Calls to these methods may only occur within an active control context. Instances of these classes may show up at the interface only in object position, i.e., as a parameter or return value but not as a callee. Additionally, we want to support the import of programming language classes. This facilitates writing a specification program, as it allows, for instance, to import standard library classes implementing common data structures as sets or lists or the like.

Table 5.4 shows the grammar for a specification language extended by programming language classes. Actually, the syntactical extension of the specification language is very similar to the modification of the last section. Again we borrow the class definition constructs from the grammar of the programming language but this time we don't have to adapt the original method definition but we keep the return expression and the type in the corresponding construct. We furthermore extend the specification construct with the support for import declarations.

The idea is to embed the class concept of the programming language in the specification language, such that classes of programming language components can

s	$::= \overline{cutdecl}\ \overline{impdecl}\ \overline{mokdecl}\ \overline{T}\,\overline{x};\ \overline{cldef}\ \{\ stmt\ \}$	specification
$cutdecl$	$::= \texttt{test class}\ C;$	test unit class
$impdecl$	$::= \texttt{import}\ C;$	imported class
$mokdecl$	$::= \texttt{mock class}\ C\{C(T,\ldots,T);\ \overline{T\ m(T,\ldots,T)}\};$	mock class
$cldef$	$::= \texttt{class}\ C\{\overline{T}\,\overline{f};\ con\ \overline{mdef}\}$	class def.
con	$::= C(\overline{T}\,\overline{x})\{\overline{T}\,\overline{x};\ stmt;\ \texttt{return}\}$	constructor
$mdef$	$::= T\ m(\overline{T}\,\overline{x})\{\overline{T}\,\overline{x};\ stmt;\ \texttt{return}\ e\}$	meth. def.
$stmt$	$::= x = e \mid x = \texttt{new}\ C() \mid \varepsilon \mid stmt; stmt \mid \{\overline{T}\,\overline{x};\ stmt\}$	statements
	$\mid \texttt{while}\ (e)\ \{stmt\} \mid \texttt{if}\ (e)\ \{stmt\}\ \texttt{else}\ \{stmt\}$	
	$\mid stmt_{in} \mid stmt_{out} \mid \texttt{case}\ \{\ \overline{stmt_{in}; stmt}\ \}$	
	$\mid f = e \mid x = e.m(e,\ldots,e) \mid x = \texttt{new}\ C(e,\ldots,e)$	
$stmt_{in}$	$::= (C\ x)?m(\overline{T}\,\overline{x}).\texttt{where}(e)\ \{\overline{T}\,\overline{x};\ stmt;\ !\texttt{return}\ e\}$	incoming stmt
	$\mid \texttt{new}(C\ x)?C(\overline{T}\,\overline{x}).\texttt{where}(e)\ \{\overline{T}\,\overline{x};\ stmt;\ !\texttt{return}\}$	
$stmt_{out}$	$::= e!m(e,\ldots,e)\ \{\overline{T}\,\overline{x};\ stmt;\ ?\texttt{return}(x).\texttt{where}(e)\}$	outgoing stmt
	$\mid \texttt{new}!C(e,\ldots,e)\ \{\overline{T}\,\overline{x};\ stmt;\ ?\texttt{return}(x).\texttt{where}(e)\}$	
e	$::= x \mid f \mid \texttt{this} \mid \texttt{null} \mid \texttt{op}(e,\ldots,e)$	expressions

Table 5.4: Extension by programming classes: syntax

be used within specifications. Thus, it is important that class definitions which represent syntactically correct and well-typed definitions regarding the programming language are also syntax valid and well-typed regarding the specification language. Moreover, such a class definition executed within a specification should gives rise to the same semantics as in the programming language. A comparison of the extended grammar of the specification language, given in Table 5.4, with the grammar of the programming language, given in Table 2.1 and Table 2.8, shows that indeed all instances of *cldef* in the grammar of Table 2.1 are also instances of *cldef* in the grammar of Table 5.4: The grammar rules for the class, constructor, and method definitions are identical; moreover, all statements of the programming language are statements of the specification language, too. The converse, however, does not hold, since the statements in the specification language also comprise the interaction statements, which do not exist in the programming language.

We want to restrict the class definitions of the specification language to class definitions of the programming language. In particular, a class' method definitions must not contain expectation statements. This restriction has to be carried out by the type system. To this end, we introduce a new kind of control context *int*, called *internal control context*, which represents a subset of the active control context. That is, every statement which is considered to occur in internal control context is also in active control context. A statement is in internal control context if the specification has the control and if the statement is also represented in the syntax of the programming language. In particular, outgoing call statements may occur in active control context but never in internal control context. Finally, in

order to restrict the class definitions to the desired ones, the type system will allow method body to contain only statements which are in internal control context.

Furthermore, the type system has to ensure that the new programming language classes do not occur as callee in specification statements. This is done by introducing another typing context, Π, which includes all typed programming language classes. Specifically, it contains the locally defined classes as well as imported classes. This way, we can distinguish programming language classes from mock classes and from the external classes that represent the component under test.

Table 5.5 shows most of the typing rules for the specification language extended with programming language classes. Rule T-Spec ensures that the classes of the component under test are included in the type context Δ and that the imported programming language classes are included in the type context Π. Furthermore, the class definitions appearing in the specification are type-checked, where the assumed local type context is extended by the global variables and where the type context for programming language classes is enriched by the defined classes themselves. Finally, regarding the same type context, also the main specification statement is checked, which again yields its control context γ.

Rule T-Class equals its pendant of the programming language apart from the necessary adaption of the judgments regarding the the new type context Π. Likewise, the rules T-Con and T-MDef resemble the corresponding programming language rules. Except for the extended assumption context, however, they additionally restrict the body statement of the method or constructor definition to statements that are well-typed in an internal control context. That is, it ensures that the statement is also a syntactical valid statement of the programming language.

As for the statements, on the one hand we introduce new rules dealing with field updates (T-VUpd$_{\mathsf{PCl}}$), method calls (T-Call$_{\mathsf{PCl}}$), and the new class instantiation statement (T-New$_{\mathsf{PCl}}$). On the other hand, the remaining typing rules regarding other statement, are borrowed from the original specification language. The new rules are again almost identical to the corresponding rules of the programming language's type system. Only the control context is added, putting the new statements in internal control context. Besides that, the type context is extended by Π, which is used to verify the correct types of a constructor or, respectively, method call's parameters. In case of a method call, it is also consulted regarding the return type. Since mock classes and classes of the component under test do not provide access to their fields, it is ensured that the three new statements indeed can only be addressed at programming language classes and their instances.

As for the remaining inherited typing rules regarding statements, they are again extended by the new name context. Some of them are also adapted regarding the control context. A block statement, for instance, may appear within a method body but also within a specification statement. The control context of a block statement's body determines also the control context of the whole block

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

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

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

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

$$[\text{T-VUpd}]\frac{\Gamma; \Delta, \Pi, \Theta \vdash e : \Gamma(x)}{\Gamma; \Delta; \Pi; \Theta \vdash x = e : \mathsf{ok}^{int}}$$

$$[\text{T-Block}]\frac{\gamma \in \{act, int\} \quad \Gamma, \overline{x}{:}\overline{T}; \Delta; \Pi; \Theta \vdash stmt : \mathsf{ok}^{\gamma}}{\Gamma; \Delta; \Pi; \Theta \vdash \{\overline{T}\ \overline{x};\ stmt\} : \mathsf{ok}^{\gamma}}$$

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

$$[\text{T-New}_{\text{PCl}}]\frac{\Gamma(x) = C \quad \Gamma; \Delta, \Pi, \Theta \vdash \overline{e} : \Pi(C)(C).dom}{\Gamma; \Delta; \Pi; \Theta \vdash x = \texttt{new}\ C(\overline{e}) : \mathsf{ok}^{int}}$$

$$[\text{T-Call}_{\text{PCl}}]\frac{\Gamma; \Delta, \Pi, \Theta \vdash e : C \quad \Gamma(x) = C \quad \Gamma; \Delta, \Pi, \Theta \vdash \overline{e} : \Pi(C)(m).dom}{\Gamma; \Delta; \Pi; \Theta \vdash x = e.m(\overline{e}) : \mathsf{ok}^{int}}$$

$$[\text{T-FUpd}_{\text{PCl}}]\frac{\Gamma; \Delta, \Pi, \Theta \vdash e : \Gamma(f)}{\Gamma; \Delta; \Theta \vdash f = e : \mathsf{ok}^{int}}$$

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

$$[\text{T-CtrlSub}]\frac{\Gamma; \Delta; \Pi; \Theta \vdash stmt : \mathsf{ok}^{int}}{\Gamma; \Delta; \Pi; \Theta \vdash stmt : \mathsf{ok}^{act}}$$

Table 5.5: Extension by programming classes: type system (stmts)

statement, which thus can be now an active or an internal control context. The instantiation of a mock class as well as incoming call and outgoing call statements are still considered as passive or, respectively, active statements which are only well-typed if the corresponding callee can be found in the commitment context Θ or the assumed test component context Δ, respectively. This way, it is assured

that a programming language class may not occur as a callee within a specification statement. Sequential composition, while-loops, and conditional statements are well-typed within any control context. However, as in the block statement case, the control context of each of these statements is determined by the control context of their sub-constituents. In particular, the conditional statement and the sequential statement are only well-typed if their sub-statements share the same control context. For the sake of brevity, we have omitted some of the rules which are transfered from the original specification language with only minor adaption as mentioned above.

Finally, we need a subsumption rule, T-CtrlSub, regarding the active and internal control context, as each internal statement, i.e., a statement which is also part of the programming language, may also occur in an active specification statement.

The grammar and the type system ensure that class definitions appearing within a well-formed specification also represent well-formed class definitions regarding the programming language. This eases the extension of the operational semantics. For, we can borrow the internal rules for internal method and constructor calls as well as for field updates from the operational semantics of the programming language without the need for any modifications. Since the extension only concerns internal computations, we do not have to extend the external transition rules.

Also the extension of the code generation is straightforward: the class definitions can be transfered to the test program without any adaption. Moreover, the newly introduced statements may only occur in active control context. In particular, they may only occur within an incoming call statement such that the code generation algorithm will let them become part of the corresponding method or constructor body. Hence, the statements can be copied into the corresponding method or constructor definition of the resulting test program.

5.3 Subtyping and inheritance

Two important concepts of object-oriented programming languages are *inheritance* and *subtyping*. The concept of inheritance facilitates the re-use of code. In the context of class-based object-oriented languages, code re-use operates on classes, i.e., one class can *inherit* the field and method definitions of another class. Subtyping refers to the concept where types are put into a partial order relation giving rise to *type compatibility*. More specifically, within a program, an expression of a certain type can be replaced by an expression of a smaller type without compromising well-typedness. Although inheritance and subtyping actually represent two different concepts, most of the mainstream class-based programming languages merge them to one concept that we will refer to as *subclassing*: A class that inherits the code from another class represents a smaller, i.e. a *subclass*, of the code-donating *superclass*. Integrating subtyping and inheritance is possible due to the fact that classes represent types.

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

Table 5.6: *Japl* with subclassing: syntax

We want to investigate the impact of introducing subclassing into our testing approach. To this end, we extend our *programming language* with single-inheritance and single-subtyping by introducing the notion of subclassing. That is, a class may have at most one superclass. In particular, we do not account for additional concepts that would introduce polymorphism, like, for instance, *Java*'s notion of interfaces. Subclasses may provide new implementations of inherited methods. In one word, we want to allow for *overriding*. To keep the extension simple, however, we restrict overriding, such that the signature of the new method definition entails exactly the same parameter and return types. In particular we do not allow covariance on return types and we do not deal with overwriting either. However, within a redefining method body we provide a keyword `super` which allows to execute the implementation of the *inherited* method definition. Finally, the extension of our language shall implement dynamic method dispatch, meaning that the method body to be executed due to a method invocation is determined not statically but at runtime.

The syntactical extension of the programming language is shown in Table 5.6. Class definitions include a reference to the superclass. Furthermore, the set of statements is extended by a method call statement and a constructor call statement addressing the method implementation of the superclass. We assume the existence of a class `Object` which provides no fields, no method definitions, and only an empty constructor body. Thus, all classes imported or defined within the program have at least class `Object` as superclass.

As for the type system, we have to incorporate the subtyping relation. To this end, we extend the type of a class with the class name of its superclass:

$$T ::= clnames \times ((MNames \cup clnames) \rightharpoonup (U \times \ldots \times U \to U))$$

Likewise, we have to adapt the auxiliary function *cltype*. Recall that *cltype* is used to extract the type of a class from its definition. We modify the original

definition, given in Section 2.2, in that the type of a class shall also include the typing information regarding its inherited methods. Thus, the function *cltype* needs to consult the typing context in order to find out the method types of the superclass:

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

We show in Table 5.7 new and, respectively, modified typing rules according to the typing rules of Table 2.2 and Table 2.9. As mentioned above, the domain of the auxiliary function *cltype* has been adapted, such that also rule T-Prog has to be changed correspondingly. Note, that a class definition might use a class as its superclass whose definition was given ahead within the program code. Thus, the commitment context is determined incrementally. Specifically, we assume that $\overline{\mathit{cldef}}$ consists of the sequence $\mathit{cldef}_1\ \mathit{cldef}_2\ \ldots\ \mathit{cldef}_n$.

The rule T-Class is merely modified in that we adapted the class definition code in the conclusion judgment. In particular we didn't change the handling of the fields. It is a crucial point that still only the fields of the defining class are incorporated into the local type context. For, the consequence is that the constructor and the method bodies do not have access to fields provided by the superclass. This way we stipulate a field access policy where all fields are considered as private in the sense that they can be accessed by instances of the defining class, only. We will see later that this decision influences the observability of some interaction.

The rules T-SupCall and T-SupNew implement the type check for the new statements, namely for the call statements that address inherited method or constructor code. Since a class type also incorporates the type information of inherited methods, we do not need to descent the type succession but we can check well-typedness directly by consulting the type of the class that contains the `super` call. To determine the class in question we only have to lookup the type of `this`. All other premises are equal to the corresponding premises of the typing rules regarding the conventional method and constructor calls. This entails a slight abuse of notation, since now the type of a class does not only consist of the method and constructor type function but it is now a pair consisting of the class name of the super class and the mentioned type function. However, we keep the notation, that is, although the type of a class C is now of the form $\Delta(C) = (D, I)$, we still write

$$\Delta(C)(m).\mathit{dom} \quad \text{and} \quad \Delta(C)(m).\mathit{ran}$$

to denote the domain and, respectively, the range of the type function I. Similarly, we write

$$\Delta(C).\mathit{supcl}$$

$$[\text{T-Prog}]\ \frac{\begin{array}{c}\Gamma' = \Gamma, \overline{x}{:}\overline{T} \\ \Theta_1 = \mathit{cltype}(\Delta, \mathit{cldef}_1)\ \ldots\ \Theta_n = \mathit{cltype}(\Delta, \Theta_{n-1}, \mathit{cldef}_n) \\ \Gamma; \Delta \vdash \overline{\mathit{impdecl}} : \mathsf{ok} \quad \Gamma'; \Delta, \Theta_n \vdash \overline{\mathit{cldef}} : \mathsf{ok} \quad \Gamma'; \Delta, \Theta_n \vdash \mathit{stmt} : \mathsf{ok}\end{array}}{\Gamma; \Delta \vdash \overline{\mathit{impdecl}};\ \overline{T}\ \overline{x};\ \mathit{cldef}_1 \ldots \mathit{cldef}_n\ \{\mathit{stmt};\ \mathtt{return}\} : \Theta_n}$$

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

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

$$[\text{T-SupNew}]\ \frac{\Gamma(\mathtt{this}) = C \qquad \Gamma; \Delta \vdash \overline{e} : \Delta(C)(C).\mathit{dom}}{\Gamma; \Delta \vdash \mathtt{super}(\overline{e}) : \mathsf{ok}}$$

Table 5.7: *Japl* with subclassing: type system (stmts)

to denote the superclass D of C.

As mentioned earlier, an instance of a class has only access to the fields that are defined in that class. This is a central aspect for the operational semantics. First of all, however, we should have a look at a small example which will reveal that the above statement is actually not completely true, if sub-classing comes into play. Consider a code fragment which consists of the definition of two classes C and D.

```
1  C extends object {
2    T x;
3    T meth1() { x = ... }
4  }
5
6  D extends C {
7    T y;
8    T meth2() { y = ...; z = super.meth1(); ... }
9  }
```

Each class definition consists of a variable declaration and a method definition. Furthermore, class D is a subclass of class C. Now assume that we have an instance o of class D and we call its method *meth2*. According to the method body of *meth2*, its execution will change the value of the variable y. This is fine, as the variable y is declared within class D. Moreover, it is true that the method body of *meth2* must not access the variable x, as it is *not* declared within the definition of D. However, *meth2* may call the method *meth1* inherited from class C. Method *meth1*, in turn, may access and even change the variable x. Therefore, although object o is an instance of class D, it may access the inherited variable x – but only by means of an invocation of an inherited method.

The consequence regarding the operational semantics is that object o is represented twice in the heap h of the program: one entry in h stores the value of x, the other entry the value of y. The idea is that one entry represents o as an object of class C and the other entry represents o as an instance of class D. Regarding the execution of method *meth2*, o can be considered as an object of class D, hence it may access the corresponding entry in h, only. Similarly, during the execution of *meth1* we may access o via the other entry only.

On account of this, we have to change the heap function in that it does not only map object names o to objects (C, F) consisting of the object's class C and the object's field values F (cf. Definition 2.3.1), but the domain of the heap is extended by class names. Thus, a heap maps pairs of object and class names to objects. That is, the set of heap functions is redefined to:

$$\mathsf{H} \stackrel{\text{def}}{=} (\mathit{CNames} \times N) \rightharpoonup \mathit{Obj}.$$

Note, that an object still is represented by a pair (C, F) consisting of the object's field function F but also of its class C. The class C is the class from which the object has been instantiated. For instance, regarding the above example object o of class D has two representations in the heap h. In particular, it is

$$h(D, o) = (D, \{y \mapsto v_x\}) \quad \text{and} \quad h(C, o) = (D, \{x \mapsto v_y\}).$$

Moreover, we introduce a new auxiliary variable `cls` which is used to determine the class of the currently executed method body. Specifically, assuming a heap h, a global variable function v, and a local variable function list μ it is

$$C = [\![\texttt{cls}]\!]_h^{\mathsf{v},\mu}$$

the class that implements the currently executed method body. Therefore, we can access the currently executed object as it is presented to the currently executed method by means of the expression $h([\![\texttt{cls}]\!]_h^{\mathsf{v},\mu}, [\![\texttt{this}]\!]_h^{\mathsf{v},\mu})$.

Apart from the field access mechanism, the above example additionally demonstrated the invocation of the inherited methods *meth1* by using the keyword `super`. Since the class type provides the name of its superclass, extending the operational semantics with the `super` calls is straightforward: instead of looking up and expanding the method body of the executing class we use the method body that is provided by the superclass.

In the example, we actually didn't need to explicitly choose the inherited method implementation by calling `super`.*meth*1() but, since D does not override *meth1*, we could have called `this`.*meth*1(), as well, getting the same result. Hence, we assume a *dynamic dispatching* of method invocations. This dispatching mechanism is realized as follows. Within a sub-class D, for each inherited method

$$T\ m(\overline{T}\ \overline{x})$$

which is not replaced by new code in terms of a new method definition, we assume an invisible method definition as follows:

$$T\ m(\overline{T}\ \overline{x})\{\ x = \texttt{super}.m(\overline{x});\ \texttt{return}(x)\ \}.$$

As for the above example, for instance, we assume a hidden method definition of the form

$$T\ meth1()\ \{\ T\ x;\ x = super.meth1();\ return(x)\ \},$$

extending the explicitly given class definition of class D. Thus, in general, the execution of a method $\texttt{this}.m(\overline{e})$ does not require a complicated look-up mechanism in order to find the class that actually implements the method. Instead, the corresponding implementation is always provided by the calling class which then might possibly call the inherited method explicitly by means of a `super` call – if it didn't override it by real user code.

Having discussed the underlying modifications of the operational semantics, let us have a look at the corresponding rules. Regarding the internal steps, we only have to change the rules of Table 2.7 that deal with field updates, internal method calls and internal object creation. Moreover, we have to add new rules for calling methods or constructors of the superclass. The rules are given in Table 5.8.

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

$$[\textsc{Call}]\ \frac{\begin{array}{c} o = [\![e]\!]_h^{\mathsf{v},\mu} \quad C = h(\_, o).class \quad \overline{T}\,\overline{x} = mparams(C,m) \quad \overline{T_l}\,\overline{x_l} = mvars(C,m) \\ \Delta \not\vdash C : [\![\ldots]\!] \quad \mathsf{v}_l = \{\texttt{cls} \mapsto C, \texttt{this} \mapsto o, \overline{x} \mapsto [\![\overline{e}]\!]_h^{\mathsf{v},\mu}, \overline{x_l} \mapsto ival(\overline{T_l})\} \end{array}}{(h, \mathsf{v}, (\mu, x = e.m(\overline{e}); mc) \circ \mathsf{CS}^b) \rightsquigarrow (h, \mathsf{v}, (\mathsf{v}_l, mbody(C,m)) \circ (\mu, \texttt{rcv}\ x;\ mc) \circ \mathsf{CS}^b)}$$

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

$$[\textsc{SupCall}]\ \frac{\begin{array}{c} o = [\![e]\!]_h^{\mathsf{v},\mu} \quad C = [\![\texttt{cls}]\!]_h^{\mathsf{v},\mu} \quad C' = \Theta(C).supcl \\ \overline{T}\,\overline{x} = mparams(C,m) \quad \overline{T_l}\,\overline{x_l} = mvars(C,m) \\ \Delta \not\vdash C' : [\![\ldots]\!] \quad \mathsf{v}_l = \{\texttt{cls} \mapsto C, \texttt{this} \mapsto o, \overline{x} \mapsto [\![\overline{e}]\!]_h^{\mathsf{v},\mu}, \overline{x_l} \mapsto ival(\overline{T_l})\} \end{array}}{\begin{array}{l} (h, \mathsf{v}, (\mu, x = \texttt{super}.m(\overline{e}); mc) \circ \mathsf{CS}^b) \rightsquigarrow \\ (h, \mathsf{v}, (\mathsf{v}_l, mbody(C,m)) \circ (\mu, \texttt{rcv}\ x;\ mc) \circ \mathsf{CS}^b) \end{array}}$$

$$[\textsc{SupNew}]\ \frac{\begin{array}{c} o = [\![e]\!]_h^{\mathsf{v},\mu} \quad C = [\![\texttt{cls}]\!]_h^{\mathsf{v},\mu} \quad C' = \Theta(C).supcl \\ \overline{T}\,\overline{x} = mparams(C,m) \quad \overline{T_l}\,\overline{x_l} = mvars(C,m) \\ \Delta \not\vdash C' : [\![\ldots]\!] \quad \mathsf{v}_l = \{\texttt{cls} \mapsto C, \texttt{this} \mapsto o, \overline{x} \mapsto [\![\overline{e}]\!]_h^{\mathsf{v},\mu}, \overline{x_l} \mapsto ival(\overline{T_l})\} \end{array}}{\begin{array}{l} (h, \mathsf{v}, (\mu, \texttt{super}(\overline{e}); mc) \circ \mathsf{CS}^b) \rightsquigarrow \\ (h, \mathsf{v}, (\mathsf{v}_l, cbody(C,m);\ \texttt{return this}) \circ (\mu, \texttt{rcv}\ \_;\ mc) \circ \mathsf{CS}^b) \end{array}}$$

Table 5.8: *Japl* with subclassing: operational semantics (int.)

Rule FUpd is modified in that we explicitly have to look up the class C whose method body is currently in execution. For, as mentioned above, the class and

program:	environment:
C2 `extends` *C1* {	*C1* `extends` *Object* {
...	...
}	}
C4 `extends` *C3* {	*C3* `extends` *C2* {
...	...
}	}

Table 5.9: Example: cross-border inheritance

the object name are needed to get the corresponding object representation (C', F) from the heap. Finally, as in the original rule, first the field function F and then, correspondingly, the heap is updated.

Regarding an internal method call, we have to find out the class from which the callee object o has been instantiated. To this end, we consult the heap regarding o. Note that for all entries of o in the heap, the yielded class is the same. The rest of the rule is quite similar to the original rule. However, the new local variable function v_l additionally stores the class C in `cls`, as its method is about to be executed. Moreover, we have to consult the assumption context Δ in order to check that C is indeed not an external class.

The modification regarding Rule New are similar to the modifications of Rule Call. Though, we do not have to look up the type C.

In order to call an inherited method via the keyword `super`, we first have to find out the caller class C via the variable `cls`. This is shown in Rule SupCall. Afterwards, we can look up C's superclass C' and, if C' is also a program class, then we can execute its method implementation as it has been explained for Rule Call, already.

Again, similar to the Rule SupCall, Rule SupNew finds out the superclass of the caller class and executes its constructor, if the superclass is a program class.

It may happen that the program extends a class of the environment or vice versa meaning that sub-class and super-class are defined on different sides. This has the effect that calls of inherited methods or constructors may cross the interface. To understand the consequences, consider the example given in Table 5.9. In the example, some environment class $C1$ is extended by a program class $C2$. Class $C2$ is again extended by an environment class $C3$ which in turn is extended by a program class $C4$. Now let us assume that an instance o of $C4$ calls an inherited method $m3$ of $C3$. This results in a cross-border method call, where the environment executes the method body of $m3$ provided by $C3$. In order to

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

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

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

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

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

potentially access fields of $C3$, object o is considered as an object of $C3$ during the execution of $m3$ as we have explained above. However, $m3$ may itself call an inherited method $m2$ of $C2$ which, in turn, may call an inherited method $m1$ of $C1$. Again, object o has to be considered as an object of $C1$ in order to access fields of $C1$. Summarizing, object o shows up twice as callee object in the environment – however, the first call needed to consider o as an object of $C3$ and the second call casted o to an object of $C1$. Therefore, regarding the external steps, we have to equip the communication labels a for method calls with a class type C of the callee object o, such that $a = \nu(\Theta').\langle \mathit{call}\ C.o.m(\overline{v})\rangle!$ and, respectively, $a = \nu(\Delta').\langle \mathit{call}\ C.o.m(\overline{v})\rangle?$. Apart from that, we only have to implement minor adaptions regarding the rules for incoming and outgoing method calls of the external semantics. The rules are given in Table 5.10.