Testing object Interactions

Grüner, A.

Appendices

Appendix A

Subject reduction

This chapter deals with the with well-typedness of configuration. We want to prove that the rules of the operational semantics preserve well-typedness of the configuration. This feature, called *subject reduction*, was formalized in Lemma 2.4.7 and what follows is the proof for this lemma. Definition 2.4.4 introduces three requirements for well-typed configurations and the idea of the proof is to make a case analysis on the transition for each requirement.

Proof. By case analysis of the transition step. As a precondition for all cases, we assume that $\Delta \vdash c : \Theta$ holds. Let h and h' be the heap functions as well as v and v' the global variable functions for the configuration c and c', respectively. Before we start with the case analysis, let us make three general observations. First, no transition rule changes the domain of the global variable function, i.e. $dom(\mathsf{v}) = dom(\mathsf{v}')$. Second, regarding external steps the new assumption-commitment context always represents an extension of the previous context. In particular, all class names in Δ and in Θ have the same type in Δ' and in Θ', respectively. Furthermore, all transition steps change the local variables and code of the top-most activation records only, if at all. Thus, within the following proof we can ignore the tail of the call stack and focus on the top-most activation records.

Now let us prove the first requirement of Definition 2.4.4, i.e., we want to show that all objects on the heap of configuration c' belong to a program class mentioned in Θ'.

Case Let us assume that $c \rightsquigarrow_p c'$.
Regarding the Rules Ass, Call, BlkBeg, BlkEnd, While$_i$, Cond$_i$, and Ret there is no change of the heap involved. As Θ' is an extension of Θ compliance with the first requirement results from the precondition.

Subcase Rule FUpd
Lets assume that $c \rightsquigarrow c'$ due to a field update. In particular, the third premise of Rule FUpd implements the actual update. It also shows, however, that the class name of the involved object is not changed. Thus, a field update does not break the requirement.

Subcase Rule New
Assume that c evolves to c' due to application of Rule New. Then the heap is extended by a new object o of class C. Likewise, the stack is extended by the method body of C. Since the auxiliary function *cbody* is only defined for program classes and as the program p is well-typed, we can deduce that $\Theta' \vdash C : [\![\ldots]\!]$.

Case Let us now assume that $\Delta \vdash c : \Theta \xrightarrow{a}_p \Delta' \vdash c' : \Theta'$.
Only one rule of the external semantics changes the heap, namely Rule NewI. Since Θ' is an extension of Θ the requirement follows from the precondition for all the other external rules. Regarding NewI, as in Rule New, we can basically deduce from the definedness of *cbody* for class name C that the first requirement of a well-typed configuration also holds for the new configuration with the extended heap. Now let us prove the second requirement of Definition 2.4.4. That is, we have to show that every free variable of each activation record of c' is a global variable or in the domain of the record's local variable list.

Case Again consider $c \rightsquigarrow_p c'$
We show the most interesting cases.

Subcase Rule Ass
Execution of the assignment statement $x = e$ does not extend the set of free variables of the corresponding activation record but instead possibly reduces it by x and *fvars*(e). Moreover, the domain of the record's local variable list is not changed which yields the proof for the requirement.

Subcase Rule Call and Rule New
Transitions that represent an internal method call or object instantiation create a new top most activation record, while the method or constructor call in the previously top most record is replaced by a receive statement. Thus, regarding the previously top most record, all free variables of the record's code are part of the record's local variable list. As for the new activation record, the code is instantiated by the method or constructor body of the corresponding program class. We know that the program is well-typed, therefore the code might only make references to global variables, to `this`, or to local variables of the method itself. Since the new record is equipped with a local variable function that consists of a mapping for the aforementioned variables, the requirement is fulfilled.

Subcase Rule Ret
An application of Rule Ret causes the removal of the top most activation record. Apart from this, only the receive statement on top of the calling activation record is removed. Thus, again all free variables of the new top most activation record are in the record's local variable list.

Case Assume $\Delta \vdash c : \Theta \xrightarrow{a}_p \Delta' \vdash c' : \Theta'$

Subcase RulesCallO and NewO
In both cases the outgoing method or constructor call is replaced by an annotated receive statement. No introduction of new variables and no modification of the

record's local variable functions is involved in this step. Thus the requirement follows from the precondition.

Subcase Rule RetO
Only the top most activation record is removed. The requirement follows from the precondition.

Subcase Rules CallI and NewI
Both rules extend the call stack by a new activation record leaving the rest of the call stack unchanged. Like in the case for internal method calls we can deduce from the well-typedness of the program that the new activation record conforms to the second requirement of the well-typedness definition for configurations.

Subcase Rule RetI
An incoming return leads to the removal of the receive statement on top of the top most activation record. Again, no new free variables are introduced and the domain of the local variable function list is not changed. Finally, we have to prove that also the third requirement for well-typed configurations is fulfilled by the new configuration c'. More specifically, we have to show that each of the call stack's activation records that represents a method or constructor execution provides a valid value for the special name `this`. Obviously, the only interesting cases are the transitions that deal with internal or incoming method and constructor calls. All other transitions do not modify the value of `this` within the local variable lists.

Case Internal step

Subcase Rule Call
The local variable function for the new activation record maps `this` to o. Moreover, the second premise of the rule verifies that o indeed is on the heap.

Subcase Rule New
In Rule New also `this` is mapped to o. In the object creation case, however, the object o is created and the new heap is extended by the new object.

Case External step

Subcase Rule CallI
The argumentation for the incoming method call is almost identical to the proof for internal method calls. The first premise of the label check T-CallI verifies that the callee object name o represents an object that is committed by the program. Furthermore, the local variable function of the new activation record maps `this` to o.

Subcase Rule NewI
Similar to the internal object creation, we can see in Rule NewI that the heap is extended with a new object referenced by o which in turn serves as the value for `this` in the local variable function. □

Appendix B

Compositionality

The goal of this section is to prove the compositionality-Lemma 2.5.5 of Section 2.5. This is structured as follows. We start with the discussion of some general features of the language's transition semantics. Afterwards we will provide a merge definition that meets the requirements of the merge function definition given in Lemma 2.5.5. This is followed by a few small proofs of some simple yet useful features of the merge function in general. The compositionality-Lemma states that the order regarding the application of the merge function on configurations, on the one hand, and application of the transition rules, on the other hand, does not play a role. Thus, the lemma consists of two directions: one direction states that regarding the transition semantics the composition of two components evolves to the same result as the two original components. The other direction says that two constituents of one (closed) program evolve to the same result as the original program. Correspondingly, the proof of Lemma 2.5.5 actually consists of two parts. First, we will show certain features about the composition of two components. Then, we show the features about the constituents of a closed program. Both cases, however, consist of several smaller sub-proofs, but the schema for both parts is the same. That is, regarding the composition we first prove the features for single internal and single external steps. Then the compositionality part follows from this by induction on the length of the trace. Similarly, regarding the decomposition we show that a single internal step of a closed program corresponds to internal or external single steps with regards to its constituents. Again, the decompositionality direction follows by induction on the length of the trace.

We begin with three small lemmas about the independence of internal deductions from certain changes regarding the stack, heap, global variables, or the component code. More specifically, the first lemma states that a single internal deduction step does only depend on the topmost but not on the trailing activation records of the call stack.

Lemma B.0.1 (Stack tail does not influence internal steps)**:** Assume two configurations

$$(h, \mathsf{v}, \mathsf{CS} \circ \mathsf{CS}_1^b), (h, \mathsf{v}, \mathsf{CS} \circ \mathsf{CS}_2^b) \in \mathit{Conf}.$$

If $(h, \mathsf{v}, \mathsf{CS} \circ \mathsf{CS}_1^b) \rightsquigarrow (h', \mathsf{v}', \acute{\mathsf{CS}} \circ \mathsf{CS}_1^b)$ then also $(h, \mathsf{v}, \mathsf{CS} \circ \mathsf{CS}_2^b) \rightsquigarrow (h', \mathsf{v}', \acute{\mathsf{CS}} \circ \mathsf{CS}_2^b)$.

Proof. By case analysis on the computation step. As for simple computation steps, i.e., computation steps which do only modify the top most activation record, the lemma follows immediately from the corresponding rules of the internal operational semantics, which are Ass, FUpd, BlkBeg, BlkEnd, Whl$_i$, and Cond$_i$. The remaining internal rules, Call, New, and Ret, deserve a closer look, as they also change the number of activation records within the call stack.

Case Rule Call
In case of an internal method call we can assume that

$$\mathsf{CS} = (\mu, x = e.m(\overline{e}); mc)$$

and correspondingly that

$$\acute{\mathsf{CS}} = (\mathsf{v}_l, \mathit{mbody}(C, m)) \circ (\mu, \mathtt{rcv}x; mc) \ .$$

Now it is easy to see that the application of Rule Call is independent of the call stack tail CS_1^b and CS_2^b, respectively.

Case Rule New
Similar to internal method calls, regarding internal constructor calls we can assume that

$$\mathsf{CS} = (\mu, x = \mathtt{new}\ C(\overline{e}); mc)$$

and correspondingly that

$$\acute{\mathsf{CS}} = (\mathsf{v}_l, \mathit{cbody}(C)) \circ (\mu, \mathtt{rcv}x; mc) \ .$$

Again, Rule New is formulated independently of the call stack tail CS_1^b and CS_2^b, respectively.

Case Rule Ret
As for an internal method or constructor return, we can define

$$\mathsf{CS} = (\mu_1, \mathtt{return}\ e) \circ (\mu_2, \mathtt{rcv}\ x; mc)$$

and

$$\acute{\mathsf{CS}} = (\mu_2', mc) \ .$$

Yet again, this definition makes the independence of Rule Ret regarding the call stack tail apparent. □

Similarly, extensions of the heap or of the global variable function do not influence the outcome of internal computation steps. This is formalized in the next lemma. For two functions f_1 and f_2 with $\mathit{dom}(f_1) \perp \mathit{dom}(f_2)$ we use the notion $f_1 \frown f_2$ for the function that represents the disjunct union of f_1 and f_2.

Lemma B.0.2 (Heap and variable extension do not affect internal steps)**:** If $(h_1, \mathsf{v}_1, \mathsf{CS}) \rightsquigarrow (h'_1, \mathsf{v}'_1, \mathsf{CS}')$ such that $dom(h'_1) \perp dom(h_2)$ then also

$$(h_1 {}^\frown h_2, \mathsf{v}_1 {}^\frown \mathsf{v}_2, \mathsf{CS}) \rightsquigarrow (h'_1 {}^\frown h_2, \mathsf{v}'_1 {}^\frown \mathsf{v}_2, \mathsf{CS}') \; .$$

Proof. Applicability of the internal transition $(h, \mathsf{v}, \mathsf{CS}) \rightsquigarrow (h', \mathsf{v}', \mathsf{CS}')$ ensures that the deduction step does not realize a call to an external class or object and that only evaluation of local variables defined in CS, of global variables of v, or object names of h might be involved. Disjunction of h'_1 and h_2 is required in order to prevent name clashes due to internal object creation. This, however, does not represent a real restriction, since we consider the semantics modulo renaming anyway, as we have remarked in 2.4.6 already. □

Also extending the program by another component does not affect the outcome of an internal step.

Lemma B.0.3 (Additional classes do not affect internal steps)**:** Assume two components p and p' such that $p \Supset p'$ is defined. If $(h, \mathsf{v}, \mathsf{CS}) \rightsquigarrow_p (h', \mathsf{v}', \mathsf{CS}')$ then also $(h, \mathsf{v}, \mathsf{CS}) \rightsquigarrow_{p \Supset p'} (h', \mathsf{v}', \mathsf{CS}')$.

Proof. Trivial, as the reduction step does only refer to method code of p, if at all. And the component merge does not modify method code of p. □

Now its time to give a concrete definition of a merge function. This merge function will form the basis of the compositionality proof.

Definition B.0.4 (Merge of configurations)**:** Given two configurations

$$(h_1, \mathsf{v}_1, \mathsf{CS}_1), (h_2, \mathsf{v}_2, \mathsf{CS}_2) \in \mathit{Conf}$$

with $\Delta \vdash (h_1, \mathsf{v}_1, \mathsf{CS}_1) : \Theta$ and $\Theta \vdash (h_2, \mathsf{v}_2, \mathsf{CS}_2) : \Delta$. We assume that $dom(h_1) \perp dom(h_2)$ as well as $dom(\mathsf{v}_1) \perp dom(\mathsf{v}_2)$ – otherwise we assume a proper renaming of objects or, respectively, variables. The result of the merge

$$(h, \mathsf{v}, \mathsf{CS}) = (h_1, \mathsf{v}_1, \mathsf{CS}_1) \Supset (h_2, \mathsf{v}_2, \mathsf{CS}_2)$$

is defined by:

- $h \stackrel{\text{def}}{=} h_1 {}^\frown h_2$,
- $v \stackrel{\text{def}}{=} v_1 {}^\frown v_2$, and
- $\mathsf{CS} \stackrel{\text{def}}{=} \mathsf{CS}_1 \mathbin{/\!\!\backslash\!\!\backslash} \mathsf{CS}_2$, where $\mathbin{/\!\!\backslash\!\!\backslash}$ denotes a commutative operation representing the merge of the two call stacks which is inductively defined by the following equations:

$$\begin{aligned}
(\mathsf{AR}^{i} \circ \mathsf{AR}^{ib} \circ \mathsf{CS}_1^{b}) \mathbin{/\!\!\backslash\!\!\backslash} \mathsf{CS}_2^{eb} &\stackrel{\text{def}}{=} \mathsf{AR}^{i} \circ (\mathsf{AR}^{ib} \circ \mathsf{CS}_1^{b}) \mathbin{/\!\!\backslash\!\!\backslash} \mathsf{CS}_2^{eb} && \text{(B.1)}\\
(\mathsf{AR}^{i} \circ \mathsf{CS}_1^{eb}) \mathbin{/\!\!\backslash\!\!\backslash} (\mathsf{AR}_2^{eb} \circ \mathsf{CS}_2^{b}) &\stackrel{\text{def}}{=} \mathsf{AR}^{i} \circ \mathsf{CS}_1^{eb} \mathbin{/\!\!\backslash\!\!\backslash} (\mathsf{AR}_2^{ib} \circ \mathsf{CS}_2^{b}) && \text{(B.2)}\\
\mathsf{AR}^{i} \mathbin{/\!\!\backslash\!\!\backslash} (\mathsf{AR}_2^{eb} \circ \mathsf{CS}_2^{b}) &\stackrel{\text{def}}{=} \mathsf{AR}^{i} \circ (\mathsf{AR}_2^{ib} \circ \mathsf{CS}_2^{b}) && \text{(B.3)}\\
\mathsf{AR}^{i} \circ \mathsf{CS}_1^{b} \mathbin{/\!\!\backslash\!\!\backslash} \epsilon &\stackrel{\text{def}}{=} \mathsf{AR}^{i} \circ \mathsf{CS}_1^{b} && \text{(B.4)}
\end{aligned}$$

Note that in AR_2^{ib} denotes the activation record that results from AR_2^{eb} by forgetting the return type of the topmost `rcv` statement.

Remark B.0.5: The equations in Definition B.0.4 show that a merge of two call stacks is only defined if exactly one call stack has an active or internally blocked activation record on top and the other call stack is externally blocked.

The next lemma makes a statement about the merge of call stacks.

Lemma B.0.6 (Topmost activation record remains topmost)**:** There exists a function f such that for all defined merges of call stacks the following holds:

1. $(\mathsf{AR}^i \circ \mathsf{CS}_1^b) \mathbin{\wedge\!\!\!\wedge} \mathsf{CS}_2^b = \mathsf{AR}^i \circ f(\mathsf{CS}_1^b, \mathsf{CS}_2^b)$.

2. In particular, the activation record that is on top of the active call stack before the merge also remains the topmost record of the resulting call stack after the merge. Moreover, the form of the rest of the resulting call stack does not depend on the topmost record but is determined only by the rest of the first stack frame and the second stack frame.

Proof. Let the function f be defined by

$$f(\mathsf{CS}_1, \mathsf{CS}_2) \stackrel{\text{def}}{=} \begin{cases} (\mathsf{AR}_1^{eb} \circ \mathsf{CS}_1^b) \mathbin{\wedge\!\!\!\wedge} (\mathsf{AR}_2^{ib} \circ \mathsf{CS}_2^b) & \text{if } \mathsf{CS}_1 = \mathsf{AR}_1^{eb} \circ \mathsf{CS}_1^b \text{ and} \\ & \phantom{\text{if }} \mathsf{CS}_2 = \mathsf{AR}_2^{eb} \circ \mathsf{CS}_2^b \\ \mathsf{CS}_1 \mathbin{\wedge\!\!\!\wedge} \mathsf{CS}_2 & \text{else} \end{cases}$$

where AR_2^{ib} represents the activation record which results from AR_2^{eb} by forgetting the type annotation of the receive statement. Then f has the property stated in the first statement. The second statement follows immediately from the definition of the merge of two stack frames. □

Now we want to apply the new lemmas in order to show that a simple internal computation step of one configuration will not be influenced if we merge it with another configuration. This is formalized in the following lemma.

Lemma B.0.7 (Merge does not influence simple deduction)**:** Assume a configuration $(h_1, \mathsf{v}_1, \mathsf{AR}^a \circ \mathsf{CS}^b)$ such that

$$(h_1, \mathsf{v}_1, \mathsf{AR}^a \circ \mathsf{CS}^b) \rightsquigarrow (h_1', \mathsf{v}_1', \mathsf{A\acute{R}}^a \circ \mathsf{CS}^b)$$

represents a simple deduction. Then, if for some other configuration $(h_2, \mathsf{v}_2, \mathsf{CS}_2^b)$ the merge $(h_1, \mathsf{v}_1, \mathsf{AR}^a \circ \mathsf{CS}^b) \mathbin{\supset\mkern-13mu{\scriptscriptstyle +}} (h_2, \mathsf{v}_2, \mathsf{CS}_2^b)$ is defined, we get

$$(h_1, \mathsf{v}_1, \mathsf{AR}^a \circ \mathsf{CS}^b) \mathbin{\supset\mkern-13mu{\scriptscriptstyle +}} (h_2, \mathsf{v}_2, \mathsf{CS}_2^b) \rightsquigarrow (h_1', \mathsf{v}_1', \mathsf{A\acute{R}}^a \circ \mathsf{CS}^b) \mathbin{\supset\mkern-13mu{\scriptscriptstyle +}} (h_2, \mathsf{v}_2, \mathsf{CS}_2^b).$$

Proof. Let us assume that

$$(h_1, \mathsf{v}_1, \mathsf{AR}^a \circ \mathsf{CS}^b) \rightsquigarrow (h_1', \mathsf{v}_1', \mathsf{A\acute{R}}^a \circ \mathsf{CS}^b).$$

We know from Lemma B.0.6 that $\mathsf{AR}^a \circ \mathsf{CS}^b \mathbin{\wedge\!\!\wedge} \mathsf{CS}_2^b = \mathsf{AR}^a \circ f(\mathsf{CS}^b, \mathsf{CS}_2^b)$. From Lemma B.0.1 and Lemma B.0.2 we can deduce

$$\begin{aligned}&(h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2, \mathsf{AR}^a \circ f(\mathsf{CS}^b, \mathsf{CS}_2^b)) \rightsquigarrow \\ &\quad (h_1'{}^\frown h_2, \mathsf{v}_1'{}^\frown \mathsf{v}_2, \acute{\mathsf{AR}}^a \circ f(\mathsf{CS}^b, \mathsf{CS}_2^b)) = (h_1', \mathsf{v}_1', \acute{\mathsf{AR}}^a \circ \mathsf{CS}^b) \mathbin{\supset\!\!\!\!+} (h_2, \mathsf{v}_2, \mathsf{CS}_2^b).\end{aligned}$$

□

Note that we didn't index the transition arrow in the previous lemma, as the lemma is independent of a certain program code. However, we certainly assume that all transitions in the lemma are understood in the context of the same program.

The next two lemmas will show one of the compositionality properties for single steps of the operational semantics. More specifically, Lemma B.0.8 states that for internal computation steps the order regarding merge operation application and transition rule application does not matter. Afterwards Lemma B.0.9 will show the same property for external computation steps.

Lemma B.0.8 ($\mathbin{\supset\!\!\!\!+}$ and $\rightsquigarrow$)**:** For two configurations $c_1, c_2 \in \mathit{Conf}$ and two component p_1 and p_2 such that $c_1 \mathbin{\supset\!\!\!\!+} c_2$ and $p_1 \mathbin{\supset\!\!\!\!+} p_2$ is defined, the following holds: If $c_1 \rightsquigarrow_{p_1} c_1'$ then $c_1 \mathbin{\supset\!\!\!\!+} c_2 \rightsquigarrow_{p_1 \mathbin{\supset\!\!\!\!+} p_2} c_1' \mathbin{\supset\!\!\!\!+} c_2$.

Proof. For simple computation steps the property has been proven by Lemma B.0.7 already. It remains to show the property also for the other internal transition rules given in Table 2.7. Let $c_1 = (h_1, \mathsf{v}_1, (\mathsf{AR}^a \circ \mathsf{CS}_1^b))$ and $c_2 = (h_2, \mathsf{v}_2, \mathsf{CS}_2^b)$.

Case Rule Ret
Applicability of Rule Ret for c_1 implies

$$c_1 = (h_1, \mathsf{v}_1, (\mu, \mathtt{return}\ e) \circ (\mu', \mathtt{rcv}\ x;\ mc) \circ \mathsf{CS}^b) \rightsquigarrow (h_1, \mathsf{v}_1', (\mu'', mc) \circ \mathsf{CS}^b).$$

Moreover, applying Equation B.1 twice as well as rule Ret, Lemma B.0.2, and Lemma B.0.1 yields

$$\begin{aligned}&c_1 \mathbin{\supset\!\!\!\!+} c_2 = (h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2, (\mu, \mathtt{return}\ e) \circ (\mu', \mathtt{rcv}\ x;\ mc) \circ (\mathsf{CS}^b \mathbin{\wedge\!\!\wedge} \mathsf{CS}_2^b)) \rightsquigarrow \\ &\quad (h_1{}^\frown h_2, \mathsf{v}_1'{}^\frown \mathsf{v}_2, (\mu'', mc) \circ (\mathsf{CS}^b \mathbin{\wedge\!\!\wedge} \mathsf{CS}_2^b)).\end{aligned}$$

On the other hand Equation B.1 yields

$$(h_1, \mathsf{v}_1', (\mu'', mc) \circ \mathsf{CS}^b) \mathbin{\supset\!\!\!\!+} c_2 = (h_1{}^\frown h_2, \mathsf{v}_1'{}^\frown \mathsf{v}_2, (\mu'', mc) \circ (\mathsf{CS}^b \mathbin{\wedge\!\!\wedge} \mathsf{CS}_2^b)).$$

Case Rule Call
Applicability of Rule Ret for c_1 implies

$$c_1 = (h_1, \mathsf{v}_1, (\mu, x = e.m(\overline{e}); mc) \circ \mathsf{CS}_1^b) \rightsquigarrow (h_1, \mathsf{v}_1, \mathsf{AR}_m^a \circ (\mu, \mathtt{rcv}\ x; mc) \circ \mathsf{CS}_1^b),$$

where AR^a_m represents the activation record that comprises the method body of the called method m. Again, by applying Equation B.1, rule CALL, Lemma B.0.2, and Lemma B.0.1 we get

$$c_1 ⫀ c_2 = (h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2, (\mu, x = e.m(\overline{e}); mc) \circ (\mathsf{CS}^b_1 ⩕ \mathsf{CS}^b_2) \leadsto \\ (h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2, \mathsf{AR}^a_m \circ (\mu, \mathtt{rcv}\ x; mc) \circ (\mathsf{CS}^b_1 ⩕ \mathsf{CS}^b_2).$$

On the other hand, applying Equation B.1 twice yields

$$(h_1, \mathsf{v}_1, \mathsf{AR}^a_m \circ (\mu, \mathtt{rcv}\ x; mc) \circ \mathsf{CS}^b_1) ⫀ c_2 = \\ (h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2, \mathsf{AR}^a_m \circ (\mu, \mathtt{rcv}\ x; mc) \circ (\mathsf{CS}^b_1 ⩕ \mathsf{CS}^b_2).$$

Case Rule NEW
The proof is almost identical to the proof for method calls. □

Lemma B.0.9 ($⫀$ and $\xrightarrow{a}$)**:** Assume two components p_1 and p_2 as well as configurations $c_1, c_2 \in \mathit{Conf}$ such that $p_1 ⫀ p_2$ and $c = c_1 ⫀ c_2$ are defined. Further, assume $\Delta \vdash c_1 : \Theta \xrightarrow{a}_{p_1} \Delta' \vdash c_1' : \Theta'$ as well as $\Theta \vdash c_2 : \Delta \xrightarrow{\bar{a}}_{p_2} \Theta' \vdash c_2' : \Delta'$. Then $c_1 ⫀ c_2 \leadsto_{p_1 ⫀ p_2} c_1' ⫀ c_2'$ as well as $c_1 ⫀ c_2 \leadsto_{p_2 ⫀ p_1} c_1' ⫀ c_2'$.

Proof. **Case** $a = \nu(\Theta').\langle \mathit{call}\ o.m(\overline{v})\rangle!$
In this case we know from rule CALLO that

$$c_1 = (h_1, \mathsf{v}_1, (\mu, x = e.m(\overline{e}); mc) \circ \mathsf{CS}^b)$$

such that $[\![e]\!]^{\mathsf{v}_1,\mu}_{h_1} = o$ and $[\![\overline{e}]\!]^{\mathsf{v}_1,\mu}_{h_1} = \overline{v}$. Moreover the rule yields

$$c_1' = (h_1, \mathsf{v}_1, (\mu, \mathtt{rcv}\ x{:}T; mc) \circ \mathsf{CS}^b)$$

On the other hand, from rule CALLI and from the complementary label $\bar{a}$ we can deduce for c_2 that

$$c_2 = (h_2, \mathsf{v}_2, \mathsf{CS}^{eb}_2) \text{ and } c_2' = (h_2, \mathsf{v}_2, \mathsf{AR}^a_m \circ \mathsf{CS}^{eb}_2).$$

It is $[\![e]\!]^{\mathsf{v}_1{}^\frown \mathsf{v}_2,\mu}_{h_1{}^\frown h_2} = [\![e]\!]^{\mathsf{v}_1,\mu}_{h_1}$ as well as $[\![\overline{e}]\!]^{\mathsf{v}_1{}^\frown \mathsf{v}_2,\mu}_{h_1{}^\frown h_2} = [\![\overline{e}]\!]^{\mathsf{v}_1,\mu}_{h_1}$. Thus, Lemma B.0.6 and Rule CALL yield

$$c_1 ⫀ c_2 = (h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2, (\mu, x = e.m(\overline{e}); mc) \circ f(\mathsf{CS}^b, \mathsf{CS}^{eb}_2)) \leadsto_{p_1 ⫀ p_2} \\ (h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2, \mathsf{AR}^a_m \circ (\mu, \mathtt{rcv}\ x; mc) \circ f(\mathsf{CS}^b, \mathsf{CS}^{eb}_2)).$$

Finally, due to Equation B.0.4 and Lemma B.0.6 we get

$$\begin{aligned} c_1' ⫀ c_2' &= (h_1, \mathsf{v}_1, (\mu, \mathtt{rcv}\ x{:}T; mc) \circ \mathsf{CS}^b) ⫀ (h_2, \mathsf{v}_2, \mathsf{AR}^a_m \circ \mathsf{CS}^{eb}_2) \\ &= (h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2, \mathsf{AR}^a_m \circ f(\mathsf{CS}^{eb}_2, (\mu, \mathtt{rcv}\ x{:}T; mc) \circ \mathsf{CS}^b)) \\ &= (h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2, \mathsf{AR}^a_m \circ (\mu, \mathtt{rcv}\ x; mc) \circ f(\mathsf{CS}^b, \mathsf{CS}^{eb}_2)). \end{aligned}$$

$\boxed{\textbf{Case}}$ $a = \nu(\Theta').\langle return(v)\rangle!$
According to rule RetO it is

$$c_1 = (h_1, \mathsf{v}_1, (\mu_1, \texttt{return } e) \circ \mathsf{CS}_1^{eb}) \text{ such that } [\![e]\!]_{h_1}^{\mathsf{v}_1,\mu_1} = v$$

and $c_1' = (h_1, \mathsf{v}_1, \mathsf{CS}_1^{eb})$. Likewise we know from rule RetI that

$$c_2 = (h_2, \mathsf{v}_2, (\mu_2, \texttt{rcv } x{:}T; mc) \circ \mathsf{CS}_2^{b}) \text{ and } c_2' = (h_2, \mathsf{v}_2', (\mu_2', mc) \circ \mathsf{CS}_2^{b}).$$

Now, due to Equation B.1, Equation B.0.4, Lemma B.0.6, and Lemma B.0.2 we get

$$\begin{aligned} c_1 \mathbin{\reflectbox{$\oplus$}} c_2 &= (h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2, (\mu_1, \texttt{return } e) \circ (\mathsf{CS}_1^{eb} \mathbin{/\!\!\backslash\!\!\backslash} (\mu_2, \texttt{rcv } x{:}T; mc) \circ \mathsf{CS}_2^{b}) \\ &= (h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2, (\mu_1, \texttt{return } e) \circ (\mu_2, \texttt{rcv } x; mc) \circ f(\mathsf{CS}_2^{b}, \mathsf{CS}_1^{eb}) \leadsto \\ & (h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2', (\mu_2', mc) \circ f(\mathsf{CS}_2^{b}, \mathsf{CS}_1^{eb}) \end{aligned}$$

On the other hand, Lemma B.0.6 yields

$$\begin{aligned} c_1' \mathbin{\reflectbox{$\oplus$}} c_2' &= (h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2', \mathsf{CS}_1^{eb} \mathbin{/\!\!\backslash\!\!\backslash} (\mu_2', mc) \circ \mathsf{CS}_2^{b}) \\ &= (h_1{}^\frown h_2, \mathsf{v}_1{}^\frown \mathsf{v}_2', (\mu_2', mc) \circ f(\mathsf{CS}_2^{b}, \mathsf{CS}_1^{eb})). \end{aligned}$$

All other cases are similar or dual. □

In the following we want to prove the other implication of the compositionality lemma. That is, we want to show that a component's sub-constituents come to the same result as the original component. However, again we first start by introducing some auxiliary lemmas. In particular the next lemma states that regarding an internal computation step one can prune the heap and the global variable function of a configuration to a minimum without influencing the outcome of the computation. More specifically, in most cases the heap can be even reduced to the object that is referenced by the variable `this` of the topmost activation record, as only field updates or field lookups of the corresponding object might be involved in the computation step. An exception is a method invocation where we also have to include the callee object into the minimal heap.

Lemma B.0.10 (Reduction of heap and variables)**:** Consider an internal computation step

$$(h, \mathsf{v}, (\mu, mc) \circ \mathsf{CS}^b) \leadsto (h', \mathsf{v}', \acute{\mathsf{CS}}^b).$$

Let v_s be the restriction of v on exactly the variables which occur in the expressions e that have been evaluated or updated due to the above mentioned computation step. Further, let $h_s = h \downarrow_{\{\mu(\texttt{this}), [\![e_c]\!]_h^{\mathsf{v},\mu}\}}$ if the computation step is a method call and e_c is the callee expression, or $h_s = h \downarrow_{\{\mu(\texttt{this})\}}$ otherwise. Then also

$$(h_s, \mathsf{v}_s, (\mu, mc) \circ \mathsf{CS}^b) \leadsto (h_s', \mathsf{v}_s', \acute{\mathsf{CS}}^b),$$

such that $h_s' = h' \downarrow_{dom(h_s')}$ and $v_s' = v' \downarrow_{dom(v_s)}$.

Proof. Straightforward. The selection process regarding the necessary objects in the heap ensures that for all possible internal transitions all objects names which might be dereferenced, leading to a lookup in the heap, are included in the minimized heap. This ensures that the minimized configuration is enabled and since the internal computations are deterministic (modulo new object names), the statement then also follows from Lemma B.0.2. Note that the final heaps h' and h'_s are equal on the complete domain of h'_s which might include a new object name due to a constructor call. □

Lemma B.0.11 (Decomposition, single step)**:** Let $c, c' \in \mathit{Conf}$ such that $c \rightsquigarrow_p c'$ for some component p. Moreover, assume name contexts Δ, Θ and components p_1 and p_2 with $p_1 \oplus p_2 = p$, $\Delta \vdash p_1 : \Theta$, and $\Theta \vdash p_2 : \Delta$ as well as configurations c_1 and c_2 with $c_1 \oplus c_2 = c$, $\Delta \vdash c_1 : \Theta$, and $\Theta \vdash c_2 : \Delta$. Then one of the following properties hold:

1. There exists a communication label a such that $\Delta \vdash c_1 : \Theta \xrightarrow{a}_{p_1} \Delta' \vdash c'_1 : \Theta'$ and $\Theta \vdash c_2 : \Delta \xrightarrow{\bar{a}}_{p_2} \Theta' \vdash c'_2 : \Delta'$ with $c'_1 \oplus c'_2 = c'$ or
2. $c_1 \rightsquigarrow_{p_1} c'_1$ such that $c'_1 \oplus c_2 = c'$ or $c_2 \rightsquigarrow_{p_2} c'_2$ such that $c_1 \oplus c'_2 = c'$.

Proof. By case analysis of the transition from c to c'. We show the most interesting cases.

$\boxed{\boxed{\textbf{Case}}}$ simple transition

That is, let $c = (h, \mathsf{v}, \mathsf{AR}^a \circ \mathsf{CS}^b) \rightsquigarrow (h', \mathsf{v}', \acute{\mathsf{AR}}^a \circ \mathsf{CS}^b)$. Then AR^a is either part of the call stack of c_1 or of c_2. Let us assume without the loss of generality that $c_1 = (h_1, \mathsf{v}_1, \mathsf{AR}^a \circ \mathsf{CS}^b_1)$. It is $\mathsf{v}_1 \subset \mathsf{v}$ and since $\Delta \vdash c_1 : \Theta$ we also know that the topmost statement of AR^a does not involve the evaluation of variables of $\mathit{dom}(\mathsf{v}) \setminus \mathit{dom}(\mathsf{v}_1)$. This fact, together with Lemma B.0.1 and Lemma B.0.10 yields $c_1 \rightsquigarrow (h'_1, \mathsf{v}'_1, \acute{\mathsf{AR}}^a \circ \mathsf{CS}^b_1)$ such that $\mathit{dom}(h'_1) = \mathit{dom}(h') \downarrow_{\mathit{dom}(h_1)}$ and $\mathit{dom}(\mathsf{v}'_1) = \mathit{dom}(\mathsf{v}') \downarrow_{\mathit{dom}(h_1)}$. This leads to $(h'_1, \mathsf{v}'_1, \acute{\mathsf{AR}}^a \circ \mathsf{CS}^b_1) \oplus c_2 = c'$.

$\boxed{\boxed{\textbf{Case}}}$ internal method call: $\mathsf{AR}^a = (\mu, e.m(\bar{e}); \mathit{mc})$

That is,

$$c = (h, \mathsf{v}, (\mu, e.m(\bar{e}); \mathit{mc}) \circ \mathsf{CS}^b) \rightsquigarrow_p (h, \mathsf{v}, \mathsf{AR}^a_m \circ (\mu, \texttt{rcv}\ x;\ \mathit{mc}) \circ \mathsf{CS}^b) = c',$$

where AR^a_m consists of the method body code of the method m. Let us assume that the calling activation record is part of c_1, i.e., $c_1 = (h_1, \mathsf{v}_1, (\mu, e.m(\bar{e}); \mathit{mc}) \circ \mathsf{CS}^b_1)$. Since c_1 is a well-typed configuration, it is $[\![e]\!]^{\mathsf{v}_1,\mu}_{h_1} = [\![e]\!]^{\mathsf{v},\mu}_{h}$ and we assume that the expression is evaluated to some object name o.

$\boxed{\textbf{Subcase}}$ $o \in \mathit{dom}(h_1)$

The precondition of the lemma regarding c_1 and p_1 as well as Lemma B.0.10 and Lemma B.0.1 yield that also

$$\begin{aligned} c_1 &= (h_1, \mathsf{v}_1, (\mu, e.m(\bar{e}); \mathit{mc}) \circ \mathsf{CS}^b_1) \rightsquigarrow_{p_1} (h_1, \mathsf{v}_1, \mathsf{AR}^a_m \circ (\mu, \texttt{rcv}\ x;\ \mathit{mc}) \circ \mathsf{CS}^b_1) \\ &= c'_1, \end{aligned}$$

Assume $c_2 = (h_2, \mathsf{v}_2, \mathsf{CS}_2^b)$. Then from $c_1 \mathbin{\ni\!\!\!\!+} c_2 = c$ and Lemma B.0.1 it follows that $\mathsf{CS}^b = f(\mathsf{CS}_1^b, \mathsf{CS}_2^b)$. And we get

$$\begin{array}{rcl} c_1' \mathbin{\ni\!\!\!\!+} c_2 & = & (h_1, \mathsf{v}_1, \mathsf{AR}_m^a \circ (\mu, \texttt{rcv}\ x;\ mc) \circ \mathsf{CS}_1^b) \mathbin{\ni\!\!\!\!+} (h_1, \mathsf{v}_2, \mathsf{CS}_2^b) \\ & = & (h_1 \frown h_2, \mathsf{v}_1 \frown \mathsf{v}_2, \mathsf{AR}_m^a \circ (\mu, \texttt{rcv}\ x;\ mc) \circ f(\mathsf{CS}_1^b, \mathsf{CS}_2^b) \\ & = & c_1 \end{array}$$

$\boxed{\textbf{Subcase}}$ $o \in dom(h_2)$

$$\begin{array}{l} \Delta \vdash c_1 : \Theta = \Delta \vdash (h_1, \mathsf{v}_1, (\mu, e.m(\overline{e}); mc) \circ \mathsf{CS}_1^b) : \Theta \xrightarrow{a}_{p_1} \\ \Delta \vdash (h_1, \mathsf{v}_1, (\mu, \texttt{rcv}\ x{:}T;\ mc) \circ \mathsf{CS}_1^b) : \Theta, \Theta' = \Delta \vdash c_1' : \Theta, \Theta', \end{array}$$

where $a = \nu(\Theta').\langle call\ o.m(\overline{v})\rangle!$. On the other hand, the stack of c_2 is externally blocked. Moreover, p and p_2 share the same class definition of the class of o such that

$$\begin{array}{l} \Theta \vdash c_2 : \Delta = \Theta \vdash (h_2, \mathsf{v}_2, \mathsf{CS}_2^{eb}) : \Delta \xrightarrow{\bar{a}}_{p_2} \\ \Theta, \Theta' \vdash (h_2, \mathsf{v}_2, \mathsf{AR}_m^a \circ \mathsf{CS}_2^{eb}) : \Delta = \Theta, \Theta' \vdash c_2' : \Delta. \end{array}$$

According to the definition of the stack merge it is

$$((\mu, \texttt{rcv}\ x{:}T;\ mc) \circ \mathsf{CS}_1^b) \mathbin{/\!\!\!\Lambda\!\!\!\backslash} (\mathsf{AR}_m^a \circ \mathsf{CS}_2^{eb}) = \mathsf{AR}_m^a \circ (\mu, \texttt{rcv}\ x;\ mc) \circ (\mathsf{CS}_1^b \mathbin{/\!\!\!\Lambda\!\!\!\backslash} \mathsf{CS}_2^{eb})$$

which proves the statement.

$\boxed{\boxed{\textbf{Case}}}$ internal return: $\mathsf{AR}^a = (\mu, \texttt{return}\ e;)$
That is,

$$c = (h, \mathsf{v}, (\mu, \texttt{return}\ e) \circ (\mu', \texttt{rcv}\ x;\ mc) \circ \mathsf{CS}^b) \rightsquigarrow_p (h, \mathsf{v}', (\mu'', mc) \circ \mathsf{CS}^b) = c',$$

Let us again assume that AR^a is part of the call stack of c_1. As for the second activation record there exist two possibilities; either it is also part of c_1 or in the call stack of c_2.

$\boxed{\textbf{Subcase}}$ receiving activation record is in c_2
Since c_1 has an active activation record on top and since $c_1 \mathbin{\ni\!\!\!\!+} c_2$ is defined, the topmost activation record of c_2 must be externally blocked. Moreover, the merge of to call stacks does not change the order of the activation records. Thus, the second activation record of c is the topmost activation record of c_2 but annotated with the return type. As a consequence for both components we get the following transitions:

$$\begin{array}{l} \Delta \vdash c_1 : \Theta = \Delta \vdash (h_1, \mathsf{v}_1, (\mu, \texttt{return}\ e) \circ \mathsf{CS}_1^{eb}) : \Theta \xrightarrow{a}_{p_1} \\ \Delta \vdash (h_1, \mathsf{v}_1, \mathsf{CS}_1^{eb}) : \Theta, \Theta' = \Delta \vdash c_1' : \Theta, \Theta' \end{array}$$

as well as

$$\begin{array}{l} \Theta \vdash c_2 : \Delta = \Theta \vdash (h_2, \mathsf{v}_2, (\mu', \texttt{rcv}\ x{:}T;\ mc) \circ \mathsf{CS}_2^b) : \Delta \xrightarrow{\bar{a}}_{p_2} \\ \Theta, \Theta' \vdash (h_2, \mathsf{v}_2', (\mu'', mc) \circ \mathsf{CS}_2^b) : \Delta = \Theta, \Theta' \vdash c_2' : \Delta, \end{array}$$

where $a = \nu(\Theta').\langle \mathit{return}(v)\rangle !$. From $c_1 \mathbin{⨮} c_2 = c$ we can deduce that $\mathsf{CS}^b = f(\mathsf{CS}^b_2, \mathsf{CS}^{eb}_1)$. Thus,

$$\mathsf{CS}^{eb}_1 \mathbin{⩕} (\mu'', mc) \circ \mathsf{CS}^b_2 = (\mu'', mc) \circ f(\mathsf{CS}^{eb}_2, \mathsf{CS}^b_2) = (\mu'', mc) \circ \mathsf{CS}^b,$$

which leads to $c'_1 \mathbin{⨮} c'_2 = c'$. Other cases are similar or dual. □

Finally, we can prove Compositionality-Lemma 2.5.5:

Proof. The proof follows directly by induction on the length of the transition sequence by applying Lemma B.0.8 and Lemma B.0.9, respectively, for the composition direction of the proof and Lemma B.0.11 for the decomposition direction. □

Appendix C

Code generation

C.1 Preprocessing

In this section, we want to show that preprocessing a specification results in a new specification such that the two specifications are behavioral equivalent regarding the interface communication. For this, as described in Section 4.4, we will provide a binary relation for which we will show that it represents a weak bisimulation. Furthermore, we will show that the pair of initial configurations of both specifications is included in the bisimulation relation. Recall, the preprocessing is basically done by means of two functions, prep_{in} and prep_{out} (cf. Table 4.2 and 4.1 in Section 4.1), which implement the preprocessing of passive and active statements, respectively. Hence the preprocessing functions are defined for *static* code, only. In order to define the bisimulation relation, we need to lift the preprocessing definition to *dynamic* code, namely to the code of activation records mc (cf. Section 3.4).

Definition C.1.1 (Preprocessed activation record code)**:** We extend range and domain of the preprocessing functions prep_{in} and prep_{out}, originally defined in Section 4.1, to

$$\mathit{prep}_{out} : mc \to mc \quad \text{and} \quad \mathit{prep}_{in} : mc \times s_{nxt} \to s_{nxt} \times mc.$$

We additionally define

$$\begin{aligned}\mathit{prep}_{out}(s^{act};\ !ret;\ mc_1^{psv}) &\overset{\text{def}}{=} \mathit{prep}_{out}(s^{act});\ !ret;\ \mathit{prep}_{in}(mc_2^{psv})\\ &\quad\text{with } (\_, mc_2^{psv}) = \mathit{prep}_{in}(mc_1^{psv}, \mathit{success})\end{aligned}$$

as well as

$$\begin{aligned}\mathit{prep}_{in}(s_1^{psv};\ x{=}?ret;\ mc^{act}, s_{nxt}) &\overset{\text{def}}{=} (s'_{nxt}, s_2^{psv};\ [i]x{=}?ret;\ \mathit{check}(i, e');\\ &\qquad \mathit{prep}_{out}(mc^{act}))\\ &\quad\text{with } (s'_{nxt}, s_2^{psv}) = \mathit{prep}_{in}(s_1^{psv}, \mathit{next} = i),\end{aligned}$$

where $!ret$ and $?ret$ abbreviate !`return`(e) and ?`return`$(T\ x')$.`where`(e), respectively.

Based on the definition above, we can define the bisimulation relation R_b. The idea is to relate each configuration of the original specification with the corresponding specification of the preprocessed specification. Thus, as for the heap and the global variables, we relate configurations which are almost identical but where the configurations of the preprocessed specification only provides the additional global variable *next* which stores an arbitrary expectation identifier i. Regarding the activation record code of configuration pairs of R_b, we basically relate code to its preprocessed variant according to the preprocessing functions of Definition C.1.1. An exception is code mc^{act} whose preprocessed variant starts with an *next* update statement s_{nxt}. For instance, the preprocessing of an outgoing call statement results in a corresponing call statement but which is preceded by an update statement. In these case we have to relate the original mc^{act} code not only to the preprocessing result but additionally to all the code that result from reducing s_{nxt} in terms of internal steps.

Definition C.1.2 (Bisimulation relation R_b)**:** We define a binary relation $R_b \subset \mathit{Conf} \times \mathit{Conf}$ over configurations of the specification language, such that for all heap functions h, global variable functions v, local variable function lists μ, and activation record code mc_1^{act} or, respectively, mc_1^{psv} exactly the following pairs are included:

1. $$((h, \mathsf{v}, (\mu, mc_1^{psv})), (h, \mathsf{v}_{+[next]}, (\mu, mc_2^{psv})))) \in R_b,$$
   if $(\_, mc_2^{psv}) = \mathit{prep}_{in}(mc_1^{psv}, \mathit{success})$.

2. $$((h, \mathsf{v}, (\mu, mc_1^{act})), (h, \mathsf{v}_{+[next]}, (\mu, mc_2^{act})))) \in R_b,$$
   $$\text{if } mc_2^{act} = \begin{cases} s'_{nxt};\ mc^{act} & \text{if } \mathit{prep}_{out}(mc_1^{act}) = s_{nxt};\ mc^{act} \text{ with} \\ & \quad (h, \mathsf{v}_{+[next]}, (\mu, s_{nxt})) \rightsquigarrow^* (h, \mathsf{v}_{+[next]}, (\mu, s'_{nxt})) \\ \mathit{prep}_{out}(mc_1^{act}) & \text{else} \end{cases}.$$

where $\mathsf{v}_{+[next]}$ represents the variable function that extends v with *next* such that *next* stores an arbitrary expectation identifier. In particular, v must not include a variable with this name, already. And correspondingly, mc_1^{act} and mc_1^{psv} must not include references to a variable *next*.

Note, according to the definition, R_b does not define a function. Instead, for each configuration c_1 with $(c_1, c_2) \in R_b$ for some configuration c_2, there exist several other configurations $c_3 \neq c_2$ such that also $(c_1, c_3) \in R_b$. For, on the one hand, the right hand side configuration may vary in the value of the global variable *next*. On the other hand, as mentioned above already, if c_1's activation record code is preprocessed resulting into code that starts with an update statement s_{nxt}, then c_1 is not only related to configurations that provide the corresponding preprocessed code but also to its successors where s_{nxt} has been reduced already.

Finally, we have to prove that the relation R_b is indeed a weak bisimulation relation. This is stated in the following lemma.

Lemma C.1.3: The binary relation R_b given in Definition C.1.2 represents a weak bisimulation in the sense of Definition 4.4.4.

Proof. Assume two configurations $c_1, c_2 \in Conf$ with $(c_1, c_2) \in R_b$. The definition of R_b implies that there exist a heap function h, a global variable function v, a local variable function list μ, and activation record code mc such that c_1 is of the form

$$c_1 = (h, \mathsf{v}, (\mu, mc))$$

and c_2 is of the form

$$c_2 = (h, \mathsf{v}_{+[next]}, (\mu, mc')),$$

where mc' corresponds to mc_2^{psv} or mc_2^{act} of Definition C.1.2. We prove the lemma by means of a case analysis regarding the construction of mc of the configuration c_1. In particular, for each case we will show both simulation directions at the same time. That is, in each case, we will prove that

- on the one hand, for each possible transition steps of c_1 to c_1'

$$\begin{array}{ccc} c_1 \rightsquigarrow c_1' & \text{implies} & c_2 \rightsquigarrow^* c_2' \\ & \text{and} & \\ \Delta \vdash c_1 : \Theta \xrightarrow{a} \Delta' \vdash c_1' : \Theta' & \text{implies} & \Delta \vdash c_2 : \Theta \overset{a}{\Longrightarrow} \Delta' \vdash c_2' : \Theta' \end{array}$$

- and, on the other hand, for each possible transition steps of c_2 to c_2'

$$\begin{array}{ccc} c_2 \rightsquigarrow c_2' & \text{implies} & c_1 \rightsquigarrow^* c_1' \\ & \text{and} & \\ \Delta \vdash c_2 : \Theta \xrightarrow{a} \Delta' \vdash c_2' : \Theta' & \text{implies} & \Delta \vdash c_1 : \Theta \overset{a}{\Longrightarrow} \Delta' \vdash c_1' : \Theta', \end{array}$$

such that in all cases $(c_1', c_2') \in R_b$. Within the proof we will refer to the firstly mentioned direction (i.e., c_2 simulates c_1) by using the right arrow ⇒ and correspondingly to the lastly mentioned direction (i.e., c_1 simulates c_2) by using the left arrow ⇐. We show some exemplary cases only as the remaining cases are similar. Note that according to the operational semantics each starting configuration only allows for either an internal or an external transition step.

Case $mc = \texttt{if}(e)\ \{s_1^{act}\}\ \texttt{else}\ \{s_2^{act}\};\ s^{act}$
In this case we have

$$mc' = \texttt{if}(e)\ \{prep_{out}(s_1^{act})\}\ \texttt{else}\ \{prep_{out}(s_2^{act})\};\ prep_{out}(s^{act})$$

according to Definition C.1.1 and to the sequential and the conditional case of Table 4.1.

Direction ⇒
We have to show that $c_1 \rightsquigarrow c_1'$ implies $c_2 \rightsquigarrow^* c_2'$, as c_1 can only be reduced by an internal transition. Specifically, the rules $\text{C}\textsc{ond}_1$ and, respectively, $\text{C}\textsc{ond}_2$

regarding the internal steps of the specification language's operational semantics yield

$$c_1 \rightsquigarrow c_1' \quad \text{with}$$
$$c_1' = (h, \mathsf{v}, (\mu, s_1^{act};\ s^{act})) \quad \text{or} \quad c_1' = (h, \mathsf{v}, (\mu, s_2^{act};\ s^{act})),$$

respectively, depending on the evaluation of $[\![e]\!]_h^{\mu,\mathsf{v}}$. Correspondingly, we get

$$\begin{aligned} &c_2 \rightsquigarrow c_2' \quad \text{with} \\ &c_2' = (h, \mathsf{v}_{+[next]}, (\mu, prep_{out}(s_1^{act});\ prep_{out}(s^{act}))) \quad \text{or} \\ &c_2' = (h, \mathsf{v}_{+[next]}, (\mu, prep_{out}(s_2^{act});\ prep_{out}(s^{act}))). \end{aligned}$$

According to the definition of $prep_{out}$ for the sequential composition, it is

$$(c_1', c_2') \in R_b.$$

$\boxed{\textbf{Direction}}$ $\Leftarrow$

Also c_2 can only be reduced by means of an internal transition, so we have to show that $c_2 \rightsquigarrow c_2'$ implies $c_1 \rightsquigarrow^* c_1'$. Again, we can only apply rule Cond$_1$ or Cond$_2$, if $[\![e]\!]_h^{\mu,\mathsf{v}_{+[next]}}$ evaluates to *true* or to *false*, respectively. Since e must not contain any references to *next*, it is

$$[\![e]\!]_h^{\mu,\mathsf{v}_{+[next]}} = [\![e]\!]_h^{\mu,\mathsf{v}}.$$

Hence, $c_1 \rightsquigarrow c_1'$ where c_1' and c_2' are of the same form as in the above proof regarding the other direction. Therefore, again, it is $(c_1', c_2') \in R_b$.

$\boxed{\textbf{Case}}$ $mc = x = e; s^{act}$

As for c_2, it is $mc' = x = e;\ prep_{out}(s^{act})$. Thus, the first statement of c_1's code and of c_2's code is the same assignment and so it is easy to see that

$$c_1 \rightsquigarrow c_1' \text{ implies that } c_2 \rightsquigarrow c_2',$$

but also conversely,

$$c_2 \rightsquigarrow c_2' \text{ implies that } c_1 \rightsquigarrow c_1',$$

such that, regarding both proof directions

$$(c_1', c_2') \in R_b.$$

$\boxed{\textbf{Case}}$ $mc = e!m(\overline{e})\ \{\ \overline{T}\,\overline{x};\ s_1^{psv};\ x = ?\texttt{return}(T\,x').\texttt{where}(e')\ \};\ s^{act}$

Then regarding the activation record code of c_2, the definition of R_b allows for the following possibilities. Either it is

$$\begin{aligned} mc' = {} & s_{nxt}';\ e!m(\overline{e})\ \{\ \overline{T}\,\overline{x};\ s_2^{psv};\ [i]\,x = ?\texttt{return}(T\,x').\texttt{where}(e')\ \}; \\ & check(i, e');\ prep_{out}(s^{act}), \end{aligned}$$

or, similarly, but without the preceding update statement, it is

$$\begin{aligned} mc' = \; & e!m(\overline{e}) \; \{ \; \overline{T}\,\overline{x}; \; s_2^{psv}; \; [i]\, x = ?\texttt{return}(T\,x').\texttt{where}(e') \; \}; \\ & check(i, e'); \; prep_{out}(s^{act}), \end{aligned}$$

with $(*)$ $(s_{nxt}, s_2^{psv}) = prep_{in}(s_1^{psv}, next = i)$ and

$$(h, \mathsf{v}_{+[next]}, (\mu, s_{nxt})) \rightsquigarrow^* (h, \mathsf{v}_{+[next]}, (\mu, s'_{nxt})).$$

$\boxed{\textbf{Direction}}$ $\Rightarrow$

The configuration c_1 can only be reduced by an outgoing method call. Therefore, for appropriate name contexts Δ, Δ', Θ and an outgoing method call label a it is

$$\Delta \vdash c_1 : \Theta \xrightarrow{a} \Delta' \vdash c'_1 : \Theta,$$

where the configuration c'_1 is of the form

$$c'_1 = (h, \mathsf{v}, (\mu', s_1^{psv}; \; x = ?\texttt{return}(T\,x').\texttt{where}(e') \; \}; \; s^{act}))$$

according to the rule CallO of the external semantics. As for c_2, if need be, we first process the update statement s'_{nxt} by internal transitions, so we get

$$\begin{aligned} c_2 \rightsquigarrow^* c'_2 = (\; h, \mathsf{v}_{+[next]}, \; & e!m(\overline{e}) \; \{ \; \overline{T}\,\overline{x}; \; s_2^{psv}; \; [i]\, x = ?\texttt{return}(T\,x').\texttt{where}(e') \; \}; \\ & check(i, e'); \; prep_{out}(s^{act}) \;), \end{aligned}$$

where the global variable function of c'_2 has only changed the value of *next*. Furthermore, the external semantics yields

$$\Delta \vdash c'_2 : \Theta \xrightarrow{a} \Delta' \vdash c''_2 : \Theta,$$

such that

$$c''_2 = (h, \mathsf{v}'_{+[next]}, s_2^{psv}; \; [i]\, x {=} ?\texttt{return}(T\,x').\texttt{where}(e'); \; check(i, e'); \; prep_{out}(s^{act})).$$

Due to the equation $(*)$ and according to Definition C.1.1 it is

$$(c'_1, c''_2) \in R_b.$$

$\boxed{\textbf{Direction}}$ $\Leftarrow$

If mc' starts with an update statement s'_{nxt} then

$$c_2 \rightsquigarrow c'_2$$

such that $(c_1, c'_2) \in R_b$. Alternatively, as shown above, the first statement of mc' can be an outgoing call statement. In this case, c_2 equals the configuration c'_2 of the other proof direction that we have discussed above already. Due to the fact, that expressions in mc' must not include references to the extra variable *next*, all outgoing call labels a, involved in a transition from c'_2 to c''_2, can also be applied to c_1 such that, again, $\Delta \vdash c_1 : \Theta \xrightarrow{a} \Delta' \vdash c'_1 : \Theta$ such that$(c'_1, c''_2) \in R_b$.

Case $mc = \texttt{if}(e)\ \{s_1^{psv}\}\ \texttt{else}\ \{s_2^{psv}\};\ s^{psv}$
According to the definition of prep_{in} in Table 4.2, it is

$$mc' = \texttt{if}(e)\ \{\tilde{s}_1^{p}\}\ \texttt{else}\ \{\tilde{s}_2^{p}\};\ \tilde{s}^{p},$$

where $(s_{nxt}, \tilde{s}^p) = \mathit{prep}_{in}(s^{psv}, \mathit{success})$ and, for each $i \in \{1, 2\}$,

$$(\_, \tilde{s}_i^p) = \mathit{prep}_{in}(s_i^{psv}, s_{nxt}).$$

Direction $\Rightarrow$
According to the operational semantics, only the internal rules $\textsc{Cond}_1$ or $\textsc{Cond}_2$ can be applied, in order to reduce the configuration c_1: if $[\![e]\!]_h^{\mu,\mathsf{v}}$ evaluates to *true* or to *false*, then $c_1 \rightsquigarrow c_1'$ such that

$$c_1' = (h, \mathsf{v}, (\mu, s_1^{psv};\ s^{psv})) \quad \text{or, resp.,} \quad c_1' = (h, \mathsf{v}, (\mu, s_2^{psv};\ s^{psv})).$$

Correspondingly, we get $c_2 \rightsquigarrow c_2'$ with

$$c_2' = (h, \mathsf{v}_{+[next]}, (\mu, \tilde{s}_1^p;\ \tilde{s}^p)) \quad \text{or, resp.,} \quad c_2' = (h, \mathsf{v}_{+[next]}, (\mu, \tilde{s}_2^p;\ \tilde{s}^p)).$$

The definition of prep_{in} regarding sequential compositions yields in both cases

$$(c_1', c_2') \in R_b.$$

Direction $\Leftarrow$
Both configurations, c_1 and c_2, can only be reduced by one of the internal rules $\textsc{Cond}_1$ or $\textsc{Cond}_2$. Moreover, recall again that e must not depend on the value of *next*. Therefore, the proof that we have given for the other direction also represents a proof for this direction.

Case $mc = (C\ x)?(\overline{T}\ \overline{x}).\texttt{where}(e)\{\overline{T_l}\ \overline{x_l};\ s^{act};\ \texttt{!return}(e')\};\ s^{psv}$
Again, according to the definition of prep_{in}, the activation record code of c_2 is

$$mc' = [i]\,(Cx)?(\overline{T}\overline{x}).\texttt{where}(e)\{\overline{T_l}\overline{x_l};\ \mathit{check}(i, e);\ \mathit{prep}_{out}(s^{act});\ s_{nxt};\ \texttt{!return}(e')\};\ \tilde{s}^p,$$

with $(s_{nxt}, \tilde{s}^p) = \mathit{prep}_{in}(s^{psv}, \mathit{success})$.

Direction $\Rightarrow$
The configuration c_1 allows for external transition steps only. In particular, it only allows transitions which are labeled with an incoming call label a such that

$$\Delta \vdash c_1 : \Theta \xrightarrow{a} \Delta' \vdash c_1' : \Theta,$$

with

$$c_1' = (h, \mathsf{v}, (\mu', s^{act};\ \texttt{!return}(e');\ s^{psv}))$$

The configuration c_2 allows for the same transition step. Specifically, it is

$$\Delta \vdash c_2 : \Theta \xrightarrow{a} \Delta' \vdash c_2' : \Theta,$$

where

$$c_2' = (h, \mathsf{v}_{+[next]}, (\mu', \mathit{check}(i,e);\ \mathit{prep}_{out}(s^{act});\ s_{nxt}; \texttt{!return}(e')\};\ \tilde{s}^p))$$

and, since we assume $\mathit{check}(i,e)$ to equal ϵ, additionally

$$c_2' \rightsquigarrow^* c_2'' = (h, \mathsf{v}_{+[next]}, (\mu', \mathit{prep}_{out}(s^{act});\ s_{nxt}; \texttt{!return}(e')\};\ \tilde{s}^p)).$$

According to the definition of prep_{in} and the definition of $\tilde{s}^p$, we get

$$(c_1, c_2'') \in R_b.$$

$\boxed{\textbf{Direction}}$ $\Leftarrow$

Also for c_2 the operational semantics permits only incoming method call steps a such that

$$\Delta \vdash c_2 : \Theta \xrightarrow{a} \Delta' \vdash c_2' : \Theta,$$

where

$$c_2' = (h, \mathsf{v}_{+[next]}, (\mu', \mathit{check}(i,e);\ \mathit{prep}_{out}(s^{act});\ s_{nxt}; \texttt{!return}(e')\};\ \tilde{s}^p)).$$

Again, equating $\mathit{check}(i,e)$ with ϵ we can further say

$$c_2' = (h, \mathsf{v}_{+[next]}, (\mu', \mathit{prep}_{out}(s^{act});\ s_{nxt}; \texttt{!return}(e')\};\ \tilde{s}^p)).$$

Finally, regarding the same name contexts and the same communication label, we get

$$\Delta \vdash c_1 : \Theta \xrightarrow{a} \Delta' \vdash c_1' : \Theta,$$

with

$$c_1' = (h, \mathsf{v}, (\mu', s^{act};\ \texttt{!return}(e');\ s^{psv})).$$

And again according to the definition of prep_{in} and the definition of $\tilde{s}^p$, we can conclude

$$(c_1, c_2'') \in R_b.$$

□

Lemma C.1.4: Assume a specification s with $\Delta \vdash s : \Theta$. Additionally, consider a specification s' that results from s by adding the global next variable and by preprocessing its main statement. Then

$$(c_{init}(s), c_{init}(s')) \in R_b.$$

Proof. Consider

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

to be a valid specification. Further, assume a specification s' such that

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

where stmt' results from either applying prep_{in} or prep_{out} to stmt, depending on the control context of the statement. Then the claim immediately follows from the Definition C.1.2 of R_b. □

C.2 Anticipation

In order to prove that the first preprocessing step indeed represents an anticipation mechanism of the expected interface communication, we first introduce some auxiliary definitions.

Definition C.2.1 (Anticipation-valid code)**:** The code mc of an activation record is said to be anticipation-valid if there exist update-statements $\grave{s}_{nxt}$ and $\acute{s}_{nxt}$ such that the judgment $\grave{s}_{nxt} \vdash_{\mathsf{as}} mc : \acute{s}_{nxt}$ is deducible according to the inference rules given in Table C.1.

Lemma C.2.2: Static anticipation-validity implies proper anticipation:

1. Assume $\grave{s}_{nxt} \vdash_{\mathsf{as}} mc^{psv} : \acute{s}_{nxt}$. Then for all heaps h, all global variable functions v, and all local variable function lists μ the following holds. If

   $$(h, \mathsf{v}, (\mu, \grave{s}_{nxt})) \rightsquigarrow^{*} (h, \mathsf{v}, (\mu, next = i))$$

   and

   $$(h, \mathsf{v}, (\mu, mc^{psv})) \rightsquigarrow^{*} (h, \mathsf{v}, (\mu, [j]\, mc^{psv}{\prime}).$$

   then $i = j$.

2. Assume $\grave{s}_{nxt} \vdash_{\mathsf{as}} mc^{act} : \acute{s}_{nxt}$. Then for all heaps h, all global variable functions v, and all local variable function lists μ the following holds. If

   $$(h, \mathsf{v}, (\mu, \grave{s}_{nxt})) \rightsquigarrow^{*} (h, \mathsf{v}, (\mu, next = i))$$

   and

   $$(h, \mathsf{v}, (\mu, mc^{act})) \xrightarrow{\gamma!} (h, \mathsf{v}, (\mu, [j]\, mc^{psv}{\prime}).$$

   then $i = j$.

Proof. Both, the passive and the active case will be proven by induction on the construction of the code. Let us first assume that

$$\grave{s}_{nxt} \vdash_{\mathsf{as}} mc^{psv} : \acute{s}_{nxt} \quad \text{and} \quad (h, \mathsf{v}, (\mu, \grave{s}_{nxt})) \rightsquigarrow^{*} (h, \mathsf{v}, (\mu, next = i))$$

for some heap h, global variable function v, and local variable function list μ. We do a case analysis regarding the code:

Case $mc^{psv} = [i]\,(C\,x)?m(\overline{T}\,\overline{x}).\mathtt{where}(e')\{\overline{T_l}\,\overline{x_l};\ s^{act};\ s_{nxt};\ !\mathtt{return}(e)\}$
According to Rule AS-CallIn it is $\grave{s}_{nxt} = next == i$. Thus trivially the proposition holds.

Case $mc^{psv} = \epsilon$
Nothing to show, as ϵ does not evolve to an incoming call or incoming return statement.

Case $mc^{psv} = s_1^{psv}; s_2^{psv}$
The proof for this case follows from the induction hypothesis and the premises $\grave{s}_{nxt} \vdash_{\mathsf{as}} s_1^{psv} : s_{nxt}$ and $s_{nxt} \vdash_{\mathsf{as}} s_2^{psv} : \acute{s}_{nxt}$ of Rule AS-Seq^{p}. However, we have to distinguish two sub-cases.

$$[\text{AS-CallIn}]\ \frac{\_ \vdash_{\mathsf{as}} s^{act} : \_ \quad s_{nxt} = \acute{s}_{nxt}}{next{=}i \vdash_{\mathsf{as}} [i]\,(C\;x)?m(\overline{T}\;\overline{x}).\mathtt{where}(e')\{\overline{T_l}\;\overline{x_l};\;\; s^{act};\;\; s_{nxt};\;\; !\mathtt{return}(e)\} : \acute{s}_{nxt}}$$

$$[\text{AS-Seq}^p]\ \frac{\grave{s}_{nxt} \vdash_{\mathsf{as}} s_1^{psv} : s_{nxt} \quad s_{nxt} \vdash_{\mathsf{as}} s_2^{psv} : \acute{s}_{nxt}}{\grave{s}_{nxt} \vdash_{\mathsf{as}} s_1^{psv}; s_2^{psv} : \acute{s}_{nxt}}$$

$$[\text{AS-While}^p]\ \frac{s_{nxt} \vdash_{\mathsf{as}} s^{psv} : \grave{s}_{nxt} \quad \grave{s}_{nxt} = \mathtt{if}(e)\ \{s_{nxt}\}\ \mathtt{else}\ \{\acute{s}_{nxt}\}}{\grave{s}_{nxt} \vdash_{\mathsf{as}} \mathtt{while}(e)\ \{s^{psv}\} : \acute{s}_{nxt}}$$

$$[\text{AS-If}^p]\ \frac{s_{nxt1} \vdash_{\mathsf{as}} s_1^{psv} : \acute{s}_{nxt} \quad s_{nxt2} \vdash_{\mathsf{as}} s_2^{psv} : \acute{s}_{nxt} \quad \grave{s}_{nxt} = \mathtt{if}(e)\ \{s_{nxt1}\}\ \mathtt{else}\ \{s_{nxt2}\}}{\grave{s}_{nxt} \vdash_{\mathsf{as}} \mathtt{if}(e)\ \{{s^{psv}}_1\}\ \mathtt{else}\ \{{s^{psv}}_2\} : \acute{s}_{nxt}}$$

$$[\text{AS-Case}]\ \frac{next = i \vdash_{\mathsf{as}} [i]\ \overline{stmt_{in}; s^{psv}} : \acute{s}_{nxt}}{next = i \vdash_{\mathsf{as}} \mathtt{case}\ [i]\ \overline{stmt_{in}; s^{psv}} : \acute{s}_{nxt}}$$

$$[\text{AS-Skip}]\ \frac{\grave{s}_{nxt} = \acute{s}_{nxt}}{\grave{s}_{nxt} \vdash_{\mathsf{as}} \epsilon : \acute{s}_{nxt}}$$

$$[\text{AS-}s^{psv}\text{-RetIn}]\ \frac{\grave{s}_{nxt} \vdash_{\mathsf{as}} s^{psv} : next = i \quad \_ \vdash_{\mathsf{as}} mc^{act} : \acute{s}_{nxt}}{\grave{s}_{nxt} \vdash_{\mathsf{as}} s^{psv};\;\; [i]\,x = ?\mathtt{return}(T\;x').\mathtt{where}(e);\;\; mc^{act} : \acute{s}_{nxt}}$$

$$[\text{AS-CallOut}]\ \frac{s_{nxt} \vdash_{\mathsf{as}} s^{psv} : next = i}{\grave{s}_{nxt} \vdash_{\mathsf{as}} s_{nxt};\;\; e!m(\overline{e})\{\overline{T_l}\;\overline{x_l}; s^{psv}; [i]\,x = ?\mathtt{return}(T\;x').\mathtt{where}(e)\} : next{=}i}$$

$$[\text{AS-Seq}^a]\ \frac{\grave{s}_{nxt} \vdash_{\mathsf{as}} s_1^{act} : s_{nxt} \quad s_{nxt} \vdash_{\mathsf{as}} s_2^{act} : \acute{s}_{nxt}}{\grave{s}_{nxt} \vdash_{\mathsf{as}} s_1^{act}; s_2^{act} : \acute{s}_{nxt}}$$

$$[\text{AS-While}^a]\ \frac{\grave{s}_{nxt} \vdash_{\mathsf{as}} s^{act} : \acute{s}_{nxt}}{\grave{s}_{nxt} \vdash_{\mathsf{as}} \mathtt{while}(e)\ \{s^{act}\} : \acute{s}_{nxt}}$$

$$[\text{AS-If}^a]\ \frac{\grave{s}_{nxt} \vdash_{\mathsf{as}} s_1^{act} : \acute{s}_{nxt} \quad \grave{s}_{nxt} \vdash_{\mathsf{as}} s_2^{act} : \acute{s}_{nxt}}{\grave{s}_{nxt} \vdash_{\mathsf{as}} \mathtt{if}(e)\ \{s_1^{act}\}\ \mathtt{else}\ \{s_2^{act}\} : \acute{s}_{nxt}}$$

$$[\text{AS-}s^{act}\text{-RetOut}]\ \frac{\grave{s}_{nxt} \vdash_{\mathsf{as}} s^{act} : \_ \quad s_{nxt} \vdash_{\mathsf{as}} mc^{psv} : \acute{s}_{nxt}}{\grave{s}_{nxt} \vdash_{\mathsf{as}} s^{act};\;\; s_{nxt};\;\; !\mathtt{return}(e); mc^{psv} : \acute{s}_{nxt}}$$

$$[\text{AS-VUpd}]\ \grave{s}_{nxt} \vdash_{\mathsf{as}} x = e : \acute{s}_{nxt}$$

Table C.1: Anticipation-valid code (static)

$\boxed{\textbf{Subcase}}$ $s_1^{psv} = \epsilon$
Then $\grave{s}_{nxt} = s_{nxt}$ and $(h, \mathsf{v}, (\mu, s_1^{psv}; s_2^{psv})) \rightsquigarrow (h, \mathsf{v}, (\mu, s_2^{psv}))$, so the proposition follows from the hypothesis regarding s_2^{psv}.

$\boxed{\textbf{Subcase}}$ $s_1^{psv} \neq \epsilon$
In this case the proposition immediately follows from the induction hypothesis regarding s_1^{psv}.

$\boxed{\boxed{\textbf{Case}}}$ $mc^{psv} = \texttt{while}(e)\ \{s^{psv}\}$
According to Rule AS-WHILE^p it is $\grave{s}_{nxt} = \texttt{if}(e)\ \{s_{nxt}\}\ \texttt{else}\ \{\acute{s}_{nxt}\}$ with s_{nxt} such that $s_{nxt} \vdash_{\mathsf{as}} s^{psv} : \grave{s}_{nxt}$. Assume that $(h, \mathsf{v}, (\mu, s_{nxt})) \rightsquigarrow^* (h, \mathsf{v}, (\mu, \mathit{next} == i)$. The hypothesis yields $(h, \mathsf{v}, (\mu, s^{psv})) \rightsquigarrow^* (h, \mathsf{v}, (\mu, [i]\, mc^{psv\prime}))$. Assume h, v, and μ such that $[\![e]\!]_h^{\mathsf{v},\mu} = \mathit{true}$. Then

$$(h, \mathsf{v}, (\mu, \grave{s}_{nxt})) \rightsquigarrow (h, \mathsf{v}, (\mu, s_{nxt})) \rightsquigarrow^* (h, \mathsf{v}, (\mu, \mathit{next} == i)).$$

as well as

$$(h, \mathsf{v}, (\mu, \texttt{while}(e)\ \{s^{psv}\})) \rightsquigarrow (h, \mathsf{v}, (\mu, s^{psv}; \texttt{while}(e)\{s^{psv}\})) \rightsquigarrow^* (h, \mathsf{v}, (\mu, [i]\, mc^{psv\prime\prime})).$$

On the other hand, now consider the case that $[\![e]\!]_h^{\mathsf{v},\mu} = \mathit{false}$. Then we get

$$(h, \mathsf{v}, (\mu, \grave{s}_{nxt})) \rightsquigarrow (h, \mathsf{v}, (\mu, s_{nxt})) \rightsquigarrow^* (h, \mathsf{v}, (\mu, \mathit{next} == i)).$$

The remaining cases are similar.

Now let us assume that

$$\grave{s}_{nxt} \vdash_{\mathsf{as}} mc^{act} : \acute{s}_{nxt} \quad \text{and} \quad (h, \mathsf{v}, (\mu, \grave{s}_{nxt})) \rightsquigarrow^* (h, \mathsf{v}, (\mu, \mathit{next} = i))$$

for some heap h, global variable function v, and local variable function list μ. We do a case analysis regarding the code:

$\boxed{\boxed{\textbf{Case}}}$ $mc^{act} = s_{nxt};\ e!m(\overline{e})\{\overline{T_l}\ \overline{x_l}; s^{psv}; [i]\, x = ?\texttt{return}(T\ x').\texttt{where}(e)\}$
Due to the premise of Rule AS-CALLOUT the proposition follows from the passive case of this lemma.

$\boxed{\boxed{\textbf{Case}}}$ $mc^{act} = s_1^{act}; s_2^{act}$
Like in the passive case we have to distinguish two sub-cases: if s_1^{act} is the empty statement or a variable update then the proposition follows from the hypothesis of the second statement. Otherwise it follows from the hypothesis of the first statement.

Again the remaining cases are straightforward.

□

While the previous deduction system checks that some code anticipates the incoming communication expectations in the context of any configuration state, the next definition in contrast captures the anticipation feature within the context of a given state.

$$[\text{AD-}s^{psv}\text{-RETI}]\ \dfrac{h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} s^{psv} : \mathit{next} = i \quad \_ \vdash_{\mathsf{as}} mc^{act} : s_{nxt}}{h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} s^{psv};\ [i]\, x = ?\mathtt{return}(T\ x').\mathtt{where}(e);\, mc^{act} : s_{nxt}}$$

$$[\text{AD-RETI}]\ \dfrac{[\![\mathit{next}]\!]_h^{\mathsf{v},\mu} = i \quad \_ \vdash_{\mathsf{as}} mc^{act} : s_{nxt}}{h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} [i]\, x = ?\mathtt{return}(T\ x').\mathtt{where}(e);\, mc^{act} : s_{nxt}}$$

$$[\text{AD-}s^{psv}]\ \dfrac{\_ \vdash_{\mathsf{as}} s^{psv} : s_{nxt} \quad [\![\mathit{next}]\!]_h^{\mathsf{v},\mu} = i \quad (h, \mathsf{v}, (\mu, s^{psv})) \rightsquigarrow^* [i]\, \mathit{stmt}_{in}}{h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} s^{psv} : s_{nxt}}$$

$$[\text{AD-}s^{act}\text{-RETOUT}]\ \dfrac{h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} s^{act} : \_ \quad s_{nxt} \vdash_{\mathsf{as}} mc^{psv} : s'_{nxt}}{h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} s^{act};\ s_{nxt};\ !\mathtt{return}(e);\ mc^{psv} : s'_{nxt}}$$

$$[\text{AD-RETOUT}]\ \dfrac{h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc^{psv} : s_{nxt}}{h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} !\mathtt{return}(e);\ mc^{psv} : s_{nxt}}$$

$$[\text{AD-}\mathit{stmt}_{out}]\ \dfrac{h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} s^{psv} : \mathit{next} = i \quad \_ \vdash_{\mathsf{as}} s^{act} : s_{nxt}}{h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} e!m(\overline{e})\{\overline{T_l}\ \overline{x_l};\, s^{psv};\, [i]\, x = ?\mathtt{return}(T\ x').\mathtt{where}(e)\};\, s^{act} : s_{nxt}}$$

$$[\text{AD-}s_{nxt}]\ \dfrac{(h, \mathsf{v}, (\mu, s'_{nxt})) \rightsquigarrow^* (h, \mathsf{v}', (\mu, \epsilon)) \quad h, \mathsf{v}', \mu \vdash_{\mathsf{ad}} s^{act} : s_{nxt}}{h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} s'_{nxt};\ s^{act} : s_{nxt}}$$

$$[\text{AD-}s^{act}]\ \dfrac{s^{act} \neq \mathit{stmt}_{out}; s_2^{act} \quad \_ \vdash_{\mathsf{as}} s^{act} : s_{nxt}}{h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} s^{act} : s_{nxt}}$$

Table C.2: Anticipation-valid configurations (dynamic)

Definition C.2.3 (Anticipation-valid configuration)**:** Assume a configuration

$$(h, \mathsf{v}, (\mu, mc)) \in \mathit{Conf}$$

of the specification language. Then we say that the configuration is *anticipation-valid*, written

$$h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc : \mathsf{anticip},$$

if the judgment $h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc : s_{nxt}$ can be derived for some update statement s_{nxt} by means of the deduction rules given in Table C.2 and Table C.1.

Lemma C.2.4 (Anticipation preprocessing establishes anticipation-validity)**:** Assume a statement stmt such that $\Delta \vdash \mathit{stmt} : \Theta$. Furthermore, for a given update statement $s_{\acute{nxt}}$ let $\mathit{stmt}' = \mathit{prep}(\mathit{stmt}, s_{\acute{nxt}})$. Then for some appropriate update statement $s_{\grave{nxt}}$ also $s_{\grave{nxt}} \vdash_{\mathsf{as}} \mathit{stmt}' : s_{\acute{nxt}}$ holds.

Proof. More specifically, we will prove in the following that, if stmt is an instance of s^{psv} then it is $s_{\grave{nxt}} \vdash_{\mathsf{as}} \mathit{stmt}' : s_{\acute{nxt}}$ with $s_{\grave{nxt}}$ defined by $(s_{\grave{nxt}}, \mathit{stmt}') = \mathit{prep}_{in}(\mathit{stmt}, s_{\acute{nxt}})$. Moreover, if stmt is an instance of s^{act} we will show that

$\_ \vdash_{\mathsf{as}} \mathit{stmt}' : \_$ holds. By structural induction. We will show the interesting subcases for both cases, i.e., passive and active statement.

$\boxed{\boxed{\textbf{Case}}}$ $\mathit{stmt} = s^{psv}$
In this case let us define $(\grave{s}_{nxt}, s^{psv\prime}) = \mathit{prep}_{in}(s^{psv}, \acute{s}_{nxt})$.

$\boxed{\textbf{Subcase}}$ $\mathit{stmt} = (C\ x)?m(\overline{T}\ \overline{x}).\mathtt{where}(e)\{\overline{T_l}\ \overline{x_l};\ s^{act};\ !\mathtt{return}(e)\}$
According to the definition of prep_{in} it is

$$\begin{array}{rcl} \grave{s}_{nxt} & = & \mathit{next} = i \quad \text{and} \\ \mathit{stmt}' & = & [i]\,(C\ x)?m(\overline{T}\ \overline{x}.\mathtt{where}(e))\{\overline{T_l}\ \overline{x_l};\ s^{act\prime};\ \acute{s}_{nxt};\ !\mathtt{return}(e)\} \quad \text{with} \\ s^{act\prime} & = & \mathit{prep}_{out}(s^{act}). \end{array}$$

The induction hypothesis implies $\_ \vdash_{\mathsf{as}} s^{act\prime} : \_$. Thus, both premises of Rule AS-CallIn are satisfied which proves the proposition.

$\boxed{\textbf{Subcase}}$ $\mathit{stmt} = \mathtt{if}(e)\ \{s_1^{psv}\}\ \mathtt{else}\ \{s_2^{psv}\}$
According to the definition of prep_{in} it is

$$\begin{array}{rcl} \grave{s}_{nxt} & = & \mathtt{if}(e)\ \{s_{nxt1}\}\ \mathtt{else}\ \{s_{nxt2}\} \text{ with} \\ (s_{nxt1}, s_1^{psv\prime}) & = & \mathit{prep}_{in}(s_1^{psv}, \acute{s}_{nxt}) \text{ and} \\ (s_{nxt2}, s_2^{psv\prime}) & = & \mathit{prep}_{in}(s_2^{psv}, \acute{s}_{nxt}). \end{array}$$

The induction hypothesis is

$$s_{nxt1} \vdash_{\mathsf{as}} s_1^{psv\prime} : \acute{s}_{nxt} \quad \text{and} \quad s_{nxt2} \vdash_{\mathsf{as}} s_2^{psv\prime} : \acute{s}_{nxt}.$$

Therefore, all three premises of Rule AS-If^p are satisfied.

$\boxed{\textbf{Subcase}}$ $\mathit{stmt} = \mathtt{while}(e)\ \{s_b^{psv}\}$
According to the definition of prep_{in} it is

$$\begin{array}{rcl} \grave{s}_{nxt} & = & \mathtt{if}(e)\ \{s_{nxt1}\}\ \mathtt{else}\ \{\acute{s}_{nxt}\} \quad \text{and} \\ \mathit{stmt}' & = & \mathtt{while}(e)\ \{s_2^{psv}\} \quad \text{with} \\ (s_{nxt1}, s_1^{psv}) & = & \mathit{prep}_{in}(s_b^{psv}, \acute{s}_{nxt}) \quad \text{and} \\ (s_{nxt2}, s_2^{psv}) & = & \mathit{prep}_{in}(s_1^{psv}, \mathtt{if}(e)\ \{s_{nxt1}\}\ \mathtt{else}\ \{\acute{s}_{nxt}\}). \end{array}$$

The induction hypothesis is

$$s_{nxt2} \vdash_{\mathsf{as}} s_2^{psv} : \mathtt{if}(e)\ \{s_{nxt1}\}\ \mathtt{else}\ \{\acute{s}_{nxt}\}.$$

Rule AS-While^p proves the claim.

$\boxed{\boxed{\textbf{Case}}}$ $\mathit{stmt} = s^{act}$

$\boxed{\textbf{Subcase}}$ $\mathit{stmt} = e!m(\overline{e})\{\overline{T_l}\ \overline{x_l};\ s^{psv};\ x = ?\mathtt{return}(T\ x').\mathtt{where}(e)\}$
According to the definition of prep_{out} it is

$$\begin{array}{rcl} \mathit{stmt}' & = & s_{nxt};\ e!m(\overline{e})\{\overline{T_l}\ \overline{x_l};\ s^{psv\prime};\ [i]\,x = ?\mathtt{return}(T\ x').\mathtt{where}(e) \quad \text{with} \\ (s_{nxt}, s^{psv\prime}) & = & \mathit{prep}_{in}(s^{psv}, \mathit{next} = i). \end{array}$$

Due to the induction hypothesis we know that $s_{nxt} \vdash_{\mathsf{as}} s^{psv\prime} : \mathit{next} = i$. This makes Rule AS-CallOut applicable which yields the proposition.

$\boxed{\textbf{Subcase}}$ $\mathit{stmt} = \texttt{while}(e)\ \{s^{act}\}$

According to the definition of prep_{out} it is

$$\mathit{stmt}' \quad = \quad \texttt{while}(e)\ \{\mathit{prep}_{in}(s^{act})\},$$

so that the induction hypothesis directly implies the proposition.

□

The next lemma justifies the term anticipation-valid configuration.

Lemma C.2.5 (Dynamic anticipation-validity implies proper anticipation)**:** Assume a configuration $(h, \mathsf{v}, (\mu, \mathit{mc})) \in \mathit{Conf}$, such that $h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} \mathit{mc} : s_{nxt}$. Then the following holds:

- If $\Delta \vdash (h, \mathsf{v}, (\mu, \mathit{mc}^{act}) \circ \mathsf{CS}) : \Theta \xrightarrow{\gamma!} \Delta \vdash (h, \mathsf{v}, (\mu, [i]\, \mathit{mc}^{psv}) \circ \mathsf{CS}) : \Theta'$ then $[\![\mathit{next}]\!]_h^{\mathsf{v},\mu} = i$.
- If $(h, \mathsf{v}, (\mu, \mathit{mc}^{psv}) \circ \mathsf{CS}) \rightsquigarrow^* (h, \mathsf{v}, (\mu, [i]\, \mathit{mc}^{psv\prime}) \circ \mathsf{CS})$ then $[\![\mathit{next}]\!]_h^{\mathsf{v},\mu} = i$.

Proof. Let us first assume that $h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} \mathit{mc}^{psv} : s_{nxt}$ and

$$(h, \mathsf{v}, (\mu, \mathit{mc}^{psv}) \circ \mathsf{CS}) \rightsquigarrow^* (h, \mathsf{v}, (\mu, [i]\, \mathit{mc}^{psv\prime}) \circ \mathsf{CS}).$$

If mc^{psv} starts with an instance of s^{psv} then the proposition immediately follows from the premises of Rule AD-s^{psv}. If mc^{psv} starts with an incoming return term then it follows immediately from the premise of Rule AD-RetI.

Now let us assume that

$$\Delta \vdash (h, \mathsf{v}, (\mu, \mathit{mc}^{act}) \circ \mathsf{CS}) : \Theta \xrightarrow{\gamma!} \Delta \vdash (h, \mathsf{v}, (\mu, [i]\, \mathit{mc}^{psv}) \circ \mathsf{CS}) : \Theta'.$$

If mc^{act} starts with an outgoing call or an outgoing return term, then the proposition follows from the passive case of this lemma. In all other cases it follows from the induction hypothesis. □

The last property that we have to show for proving Lemma 4.1.3 is that the dynamic anticipation-validity is an invariant regarding transitions of the operational semantics.

Lemma C.2.6 (Invariance of anticipation-validity)**:** Assume two specification language configurations, c and c', with

$$c = (h, \mathsf{v}, (\mu, \mathit{mc})) \quad \text{such that} \quad h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} \mathit{mc} : s_{nxt}$$

and furthermore

$$c' = (h', \mathsf{v}', (\mu', \mathit{mc}')) \quad \text{with} \quad c \rightsquigarrow c' \quad \text{or} \quad \Delta \vdash c : \Theta \xrightarrow{a} \Delta' \vdash c' : \Theta'.$$

The it is also true that

$$h', \mathsf{v}', \mu' \vdash_{\mathsf{ad}} \mathit{mc}' : s_{nxt}.$$

Proof. Case analysis regarding the construction of mc of configuration c.

$\boxed{\boxed{\textbf{Case}}}$ $mc = s^{psv};\ [i]?\texttt{return}(T\,x).\texttt{where}(e);\ mc^{act}$
We present three exemplary subcases, as the remaining cases are similar.

$\boxed{\textbf{Subcase}}$ $s^{psv} = \mathsf{if}(e)\ \{s_1^{psv}\}\ \texttt{else}\ \{s_2^{psv}\};\ s_3^{psv}$
The assumed anticipation-validity regarding c is due to Rule AD-s^{psv}-RetI, which in particular implies

$$h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} \mathsf{if}(e)\ \{s_1^{psv}\}\ \texttt{else}\ \{s_2^{psv}\};\ s_3^{psv} : \textit{next} = i. \tag{C.1}$$

According to the operational semantics, regarding c' we can conclude that $h' = h$, $\mathsf{v}' = \mathsf{v}$, and $\mu' = \mu$. Moreover, depending on the evaluation of e, the conditional statement reduces either to s_1^{psv} or s_2^{psv}. Without the loss of generality, let us assume that e evaluates to true. Thus,

$$c' = (h, \mathsf{v}, (\mu,\ s_1^{psv};\ s_3^{psv};\ [i]?\texttt{return}(T\,x).\texttt{where}(e);\ mc^{act})).$$

In order to prove $h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc' : s_{nxt}$, we have to show that

$$h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} s_1^{psv};\ s_3^{psv} : \textit{next} = i.$$

Referring to Rule AD-s^{psv}, we can see from Equation C.1 that

$$\_ \vdash_{\mathsf{as}} \mathsf{if}(e)\ \{s_1^{psv}\}\ \texttt{else}\ \{s_2^{psv}\};\ s_3^{psv} : \textit{next} = i$$

which in turn implies that also

$$\_ \vdash_{\mathsf{as}} s_1^{psv};\ s_3^{psv} : \textit{next} = i$$

due to the Rules AS-If^p and AS-Seq^p. Furthermore the premise

$$(h, \mathsf{v}, (\mu, \mathsf{if}(e)\ \{s_1^{psv}\}\ \texttt{else}\ \{s_2^{psv}\};\ s_3^{psv})) \rightsquigarrow^* (h, \mathsf{v}, (\mu, [i]\,\textit{stmt}_{in};\ s_4^{psv}))$$

of Rule AD-s^{psv} implies that also

$$(h, \mathsf{v}, (\mu, s_1^{psv}\ ;\ s_3^{psv})) \rightsquigarrow^* (h, \mathsf{v}, (\mu, [i]\,\textit{stmt}_{in};\ s_4^{psv}))$$

is true. Therefore, we get

$$h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} s_1^{psv};\ s_3^{psv} : \textit{next} = i$$

.

$\boxed{\textbf{Subcase}}$ $s^{psv} = [j]\,(C\,x)?m(\overline{T}\,\overline{x}).\texttt{where}(e)\{\ \overline{T_l}\,\overline{x_l};\ s^{act};\ !\texttt{return}(e')\ \};\ s_3^{psv}$
Similar to the previous subcase, the premise of Rule AD-s^{psv}-RetI yields

$$h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} [j]\,(C\,x)?m(\overline{T}\,\overline{x}).\texttt{where}(e)\{\ \overline{T_l}\,\overline{x_l};\ s^{act};\ !\texttt{return}(e')\ \};\ s_3^{psv} : \textit{next} = i,$$

which, according to the Rules AD-s^{psv}, AS-Seq^p, and AS-CallIn, implies that

$$[\![\textit{next}]\!]_h^{\mathsf{v},\mu} = j \quad \text{and} \quad s^{act} = s_1^{act};\ s'_{nxt} \quad \text{with} \quad s'_{nxt} \vdash_{\mathsf{as}} s_3^{psv} : \textit{next} = i.$$

The configuration c may only evolve to c' in terms of an incoming method call which leads to

$$c' = (h, \mathsf{v}, (\mathsf{v}_l \cdot \mu, s_1^{act};\ s'_{nxt}; !\mathtt{return}(e');\ s_3^{psv};\ [i]?\mathtt{return}(T\,x).\mathtt{where}(e);\ mc^{act})).$$

Thus, it remains to show that

$$h, \mathsf{v}, \mathsf{v}_l \cdot \mu \vdash_{\mathsf{ad}} s_1^{act};\ s'_{nxt};\ !\mathtt{return}(e');\ s_3^{psv} : \mathit{next} = i.$$

This, however, is true according to Rule AD-s^{act}-RetOut.

Subcase $s^{psv} = \epsilon$

In this subcase, the code mc of c starts with the outgoing call term $!\mathtt{return}(e)$, so the assumption about the anticipation-validity regarding c is due to Rule Ad-RetI. Since its premise $\_ \vdash_{\mathsf{as}} mc^{act} : s_{nxt}$ also implies anticipation-validity of mc^{act} regarding any heap and variable functions and since mc reduces to mc^{act} through an incoming return label, we can immediately see that

$$h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc^{act} : s_{nxt}.$$

Case $mc = s^{psv}$

As for configurations c whose code consist of a passive statement only, the corresponding proofs can be easily derived from the previous case. Basically, we only have to omit the trailing code $[i]\,x = ?\mathtt{return}(T\,x).\mathtt{where}(e);\ mc^{act}$.

Case $mc = s^{act};\ !\mathtt{return}(e);\ mc^{psv}$

Also regarding active code, we will show the most interesting subcases.

Subcase $s^{act} = x = e;\ s_1^{act}$

Therefore, c internally reduces to

$$c' = (h, \mathsf{v}', (\mu', s_1^{act};\ !\mathtt{return}(e);\ mc^{psv})).$$

According to Rule AD-s^{act}-RetOut it is $s_1^{act} = s_2^{act}; s'_{nxt}$ such that

$$s^{act} = x = e;\ s_2^{act};\ s'_{nxt} \quad \text{with} \quad s'_{nxt} \vdash_{\mathsf{as}} mc^{psv} : s_{nxt}.$$

We now have to distinguish the case, where x is the next variable, from the case where x represents a different variable.

Subsubcase $x \neq \mathit{next}$

As the first statement of s^{act} is not an outgoing call, but also not an instance of s_{nxt}, we know from Rule AD-s^{act} that

$$\_ \vdash_{\mathsf{as}} x = e;\ s_2^{act};\ s'_{nxt} : s'_{nxt}.$$

Consequently, it is also true that

$$\_ \vdash_{\mathsf{as}} s_2^{act};\ s'_{nxt} : s'_{nxt}.$$

This, in turn, leads to the fact that, according to Rule AD-s_{nxt}-RetOut, also

$$h, \mathsf{v}', \mu' \vdash_{\mathsf{ad}} s_2^{act};\; s'_{nxt};\; \texttt{!return}(e);\; mc^{psv}$$

is true.

$\boxed{\textit{Subsubcase}}$ $x = next$
In this case the local variable list is not changed by the internal transition, i.e., $\mu' = \mu$. Moreover, we have to consult Rule AD-s_{nxt} instead of Rule AD-s^{act}. And this rule's two premises, applied to our assignment, leads to

$$(h, \mathsf{v}, (\mu, x = e)) \rightsquigarrow (h, \mathsf{v}', (\mu, \epsilon)),$$

such that $h, \mathsf{v}', \mu \vdash_{\mathsf{ad}} s_2^{act};\; s'_{nxt} : s'_{nxt}$. Therefore, in particular the first but also the second premise of Rule AD-s^{act}-RetOut are true regarding the configuration c'.

$\boxed{\textbf{Subcase}}$ $s^{act} = e!m(\overline{e})\ \{\overline{T}\,\overline{x};\; s^{psv};\; [i]\,x = ?\texttt{return}(T\,x).\texttt{where}(e')\ \}; s_1^{act}$
Rule AD-$stmt_{out}$ yields

$$h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} s^{psv} : next = i \quad \text{and} \quad \_ \vdash_{\mathsf{as}} s_1^{act} : s_{nxt}.$$

Thus, the transition from c to c' in terms of an outgoing method call label leads to

$$c' = (h, \mathsf{v}, (\mu, s^{psv};\; [i]\,x = ?\texttt{return}(T\,x).\texttt{where}(e');\; s_1^{act};\; \texttt{!return}(e);\; mc^{psv})).$$

According to Rule AD-s^{psv}-RetI, it remains to show that

$$\_ \vdash_{\mathsf{as}} s_1^{act};\; \texttt{!return}(e);\; mc^{psv} : s_{nxt}.$$

Since we assume that c is anticipation-valid and due to Rule AD-s^{act}-RetOut it is $s_1^{act} = s_2^{act}; s'_{nxt}$ such that

$$s'_{nxt} \vdash_{\mathsf{as}} mc^{psv} : s_{nxt}.$$

Therefore, according to Rule AS-s^{act}-RetOut, it is indeed

$$\_ \vdash_{\mathsf{as}} s_2^{act};\; s'_{nxt};\; \texttt{!return}(e);\; mc^{psv} : s_{nxt}.$$

$\boxed{\textbf{Subcase}}$ $s^{act} = \epsilon$
Therefore, it is

$$h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} \texttt{!return}(e);\; mc^{psv} : s_{nxt}$$

and additionally

$$h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc^{psv} : s_{nxt}.$$

Since c evolves to

$$c' = (h, \mathsf{v}, (\mu, mc^{psv})),$$

this implies $h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc^{psv} : s_{nxt}$.

$\boxed{\boxed{\textbf{Case}}}$ $mc = s^{act}$
Much as the proof for passive statement represents a simplified case of passive call stack code, also the proof for active statements are very similar to the previous proof case. □

C.3 Correctness of the generated code

In this section we want to prove that a preprocessed specification and the correspondingly generated *Japl* code are testing bisimilar. This will also represent a proof for Lemma 3.6.2 as it stated for each specification the general existence of a program of the programming language which is “trace-equal” modulo input-enabledness. To prove testing bisimilarity, we will first define a binary relation R_t over specification language and programming language configurations. Afterwards we will prove that R_t is a testing bisimulation. Note in this section we have to deal with constructs of the specification language and, at the same time, with constructs of the programming language sharing the same name due to our language extension approach. Therefore, in the following, we will annotate constructs of the specification language with *sp* (e.g. $stmt_{sl}$) and those of the programming language with *pl* (e.g. $stmt_{pl}$). Yet we may omit the annotation in cases where the affiliation of a construct is clear.

The relation R_t is defined over configurations. However, the definition will be based on similar relations over statements and, respectively, over call stacks. Thus, before we will give the actual definition for R_t we need to define the relations regarding statements and call stacks.

Definition C.3.1: The relation $\sim_{st} \subseteq stmt_{sl} \times stmt_{pl}$ is recursively defined by the equations shown in Table C.3.

$$s^{psv} \sim_{st} \varepsilon$$
$$\texttt{if}(e)\ \{s_1^{act}\}\ \texttt{else}\ \{s_2^{act}\} \sim_{st} \texttt{if}(e)\ \{stmt_1\}\ \texttt{else}\ \{stmt_2\}$$
$$\quad \text{with } s_1^{act} \sim_{st} stmt_1 \text{ and } s_2^{act} \sim_{st} stmt_2$$
$$\texttt{while}(e)\ \{s^{act}\} \sim_{st} \texttt{while}(e)\ \{stmt\} \quad \text{with } s^{act} \sim_{st} stmt$$
$$s_1^{act};\ s_2^{act} \sim_{st} stmt_1;\ stmt_2 \quad \text{with } s_1^{act} \sim_{st} stmt_1 \text{ and } s_2^{act} \sim_{st} stmt_2$$
$$x = e \sim_{st} x = e$$
$$e!m(\overline{e})\ \{\ s^{psv}; [i]\, x = ?\texttt{return}(T\, x).\texttt{where}(e)\ \} \sim_{st} x = e.m(\overline{e});\ check(i, e)$$
$$\texttt{new}!C(\overline{e})\{\ s^{psv}; [i]\, x = ?\texttt{return}(C\, x).\texttt{where}(e)\ \} \sim_{st} x = \texttt{new}\ C(\overline{e});\ check(i, e)$$

Table C.3: Simulation relation for statements

Note, the relation $\sim_{st}$ relates all passive (specification language) statements to the empty (programming language) statement. Similarly, active method and constructor call statements of the specification language are related to the corresponding method or constructor call of the programming language, ignoring the passive statement s^{psv} that forms the body of the original call expectation statement.

Additionally, note that regarding the relation $\sim_{st}$, the expectation bodies of method and constructor calls must not provide variable declarations. Likewise, the block statement is not part of the relation. Therefore, a specification statement (as well as the corresponding program statement) of this relation never contains local variable declarations apart from the formal parameters of incoming calls.

Lemma C.3.2: Assume a preprocessed specification statement $stmt_{sl}$ and, correspondingly, a programming language statement $stmt_{pl}$ that results from generating code from $stmt_{sl}$ by means of $code_{in}$ or, respectively, $code_{out}$. Then it is $stmt_{sl} \sim_{st} stmt_{pl}$.

Proof. By structural induction. Straightforward. For instance, all passive statements are completely transcribed to *method body* code by $code_{in}$ such that no main body statement is generated at all. Similarly, all other cases immediately follow from the definition of $code_{out}$, given in Tale 4.5, and the definition of $\sim_{st}$, given in Table C.3. □

The next definition specifies a relation over activation records of the specification language and the corresponding call stack of the programming language. The definition is based on the previously defined relation over statements. However, it additionally has to consider the languages' different handling concerning the local variables. For, regarding the specification language, an incoming call results in an extension of the local variable list of the call stack's topmost (and only) activation record by a local variable functions v_l capturing the parameters of an incoming call. Within the programming language, in contrast, an incoming call causes the creation of a new activation record with its own variable function list. Moreover, while we assume that the specification does not introduce any local variables (apart from the parameter of a incoming method or constructor call), meaning that the local variable functions only consists of the formal parameters, the corresponding variable function of the programming language, in contrast, additionally provides a variable *retVal*.

Definition C.3.3: The relation $\sim_{CS} \subseteq \mathsf{AR}_{sl} \times \mathsf{CS}_{pl}$ consists of pairs of specification activation records and programming language calls stacks. It is $(\mu_{sl}, mc_{sl}) \sim_{CS} \mathsf{CS}_{pl}$ in exactly the following cases

1. $(\mathsf{v}_\perp, s^{act}) \sim_{CS} (\mathsf{v}_\perp, stmt;\ \mathtt{return})$ if $s^{act} \sim_{st} stmt$,
2. $(\mathsf{v}{\cdot}\mu, s^{act}; !\mathtt{return}(e);\ mc^{psv}) \sim_{CS} (\check{\mathsf{v}}, stmt;\ retVal = e;\ \mathtt{return}(retVal)) \circ \mathsf{CS}^{eb}$ if $s^{act} \sim_{st} stmt$ and $(\mu, mc^{psv}) \sim_{CS} \mathsf{CS}^{eb}$,
3. $(\mathsf{v}_\perp, s^{psv}) \sim_{CS} (\mathsf{v}_\perp, \varepsilon)$, and
4. $(\mathsf{v}_\perp{\cdot}\mathsf{v}{\cdot}\mu, s^{psv};\ [i]\, x = ?\mathtt{return}(T\ x).\mathtt{where}(e);\ mc^{act}) \sim_{CS}$ if $(\check{\mathsf{v}}, \mathtt{rcv}\ T{:}x;\ check(i, e);\ mc) \circ \mathsf{CS}'$ $(\mathsf{v}{\cdot}\mu, mc^{act}) \sim_{CS} (\check{\mathsf{v}}, mc) \circ \mathsf{CS}'$.

With $\check{\mathsf{v}}$ we denote the variable function that results from extending v with an additional variable *retVal*.

Before we can define the actual testing bisimulation relation R_t we have to deal with another crucial difference between a specification and a program. That is, a program provides method code which is to be copied into the program configuration at runtime, whenever a corresponding method invocation occurs. Hence, relating configurations of the specification language with configurations of the programming language is not sufficient but the static code, given in terms of method body code, has to meet certain requirements, as well. One solution would be to extend the codomain of the relation R_t such that it does not only comprise the configurations $Conf_{pl}$ of the programming language but additionally its programs p. Thus, the relation R_t would be a subset of $Conf_{sl} \times (p \times Conf_{pl})$. However, static code, as the name implies, does not change during the program execution. To express this, we choose a slightly different approach, that is, we annotate R_t^p with a specific program p, and *for each* p the relation R_t^p is a subset of $Conf_{sl} \times Conf_{pl}$. Based on this notation, we will now discuss the requirements of R_t^p that are related to the static code provided by p. This has the following three aspects.

- The program p on its own has to provide certain features which are independent of any configurations. In particular, if p does not have these features then the corresponding relation R_t^p is the empty set.

- It has been said, that static code may be copied into the program configuration in order to execute it. Executability entails the requirement that certain expressions within the code must be evaluable. This, in turn, imposes corresponding requirements on the configurations of R_t^p regarding the variable assignments given in terms of the configuration's variable functions . Thus, on the one hand, the variable functions of a configuration of R_t^p have to provide values of a proper type such that the expressions can be evaluated. On the other hand, the code of a configuration of R_t^p must not implement assignments to variables which result in a wrongly typed variable.

- Finally, the method code within p must be able to simulate all the incoming call expectations that are implemented in specification configuration of R_t^p.

In the following, we will discuss these three aspects in more detail and provide corresponding definitions. Afterwards, we will use these definitions to formulate the definition of the relation R_t^p.

First, let us deal with the general requirements regarding the static code itself. For instance, a straightforward requirement is that all methods must provide well-formed code, only. More specifically, as we have discussed in Chapter 4, the body of a method must provide a structure that allows the simulation of, not only one but potentially several, incoming call statements. To this end, we assume that the code structure follows our anticipation strategy, meaning that each method definition of p implements a case switch regarding the communication identifier and the corresponding where-clause. This requirement is formulated by the following definition.

Definition C.3.4 (Anticipation-based code structure)**:** Assume a well-typed program p. We say that p has an *anticipation-based code structure*, if for each method m of each class C of p the definition is of the following form

$$
\begin{aligned}
T\ m(\overline{T}\ \overline{x})\{\ & T\ \mathit{retVal}; \\
& \prod_{k=1}^{n} (\mathsf{if}((\mathit{next} == i_k)\ \&\&\ (e_k))\ \{\ \mathit{stmt}_k;\ \mathit{retVal} = e'_k\ \}\ \texttt{else}\ )\ \{\mathit{fail};\} \\
& \texttt{return}(\mathit{retVal})\ \}
\end{aligned}
$$

and, correspondingly, for each class C the definition of its constructor is of the following form

$$
\begin{aligned}
C\ C(\overline{T}\ \overline{x})\{\ & \mathsf{if}(\mathit{internal})\ \{\varepsilon\}\ \texttt{else} \\
& \prod_{k=1}^{n} (\mathsf{if}((\mathit{next} == i_k)\ \&\&\ (e_k))\ \{\ \mathit{stmt}_k\ \}\ \texttt{else}\ )\ \{\mathit{fail};\} \\
& \texttt{return}\ \}.
\end{aligned}
$$

We use the $\prod$ symbol to denote an iteration of nested conditional statements. Each condition expression tests for the next expected communication identifier and the corresponding where-clause. If the method invocation does not match any implemented call expectations regarding this method, then *fail* is called.

As for the relation R^p_t we will presume that p has an anticipation-based code structure. Otherwise, the relation is considered to be the empty set. Note that this requirement can be checked independently of any configurations. If p has the desired structure, however, it imposes additional requirements on the configurations of R^p_t. On the one hand, it is necessary that for each method the expressions e_1 to e_n of Definition C.3.4 can be evaluated. Since we assume that the program does not use local variables or fields (cf. code generation algorithm), this represents a requirement on the global variable function v of the configurations. In particular, v must provide defined values for all global variables that occur in e_1 to e_n. Specifically, the types of the provided values must be as assumed by the expressions, as otherwise their evaluation is not defined and the program can get stuck. Moreover, the code of a specification configuration of R^p_t must not change the type of global variables by performing a wrongly typed assignment.

Definition C.3.5 (Well-typed variable function and specification configuration)**:** Let Δ be a global and Γ a local type mapping. Further, assume a variable function list $\mu = \mathsf{v}{\cdot}\mu'$. We say, μ is *well-typed* regarding Γ and Δ, written

$$\Gamma; \Delta \vdash_{\mathsf{var}} \mathsf{v}{\cdot}\mu' : \mathsf{ok},$$

if, and only if,

$$
\begin{aligned}
\Gamma = \Gamma_1, \Gamma_2 \quad & \text{such that } \mathit{dom}(\Gamma_1) = \mathit{dom}(\mathsf{v}), \\
& \text{for all } x \in \mathit{dom}(\mathsf{v}).\ \Delta(\mathsf{v}(x)) = \Gamma_1(x), \text{ and} \\
& \Gamma_2; \Delta \vdash_{\mathsf{var}} \mu' : \mathsf{ok}.
\end{aligned}
$$

$$[\text{T-}s^{act}\text{-}\textsc{RetO}]\,\frac{\Gamma;\Delta \vdash \mathit{stmt} : \mathsf{ok}^{act} \qquad \Gamma;\Delta \vdash \mathit{mc}^{psv} : \mathsf{ok}^{psv}}{\Gamma;\Delta \vdash \mathit{stmt};\ \texttt{!return}(e);\ \mathit{mc}^{psv} : \mathsf{ok}^{psv}}$$

$$[\text{T-}s^{psv}\text{-}\textsc{RetI}]\,\frac{\Gamma;\Delta \vdash \mathit{stmt} : \mathsf{ok}^{psv} \qquad \Gamma;\Delta \vdash \mathit{mc}^{act} : \mathsf{ok}^{act}}{\Gamma;\Delta \vdash \mathit{stmt};\ ?\texttt{return}(T\,)\texttt{.where}(e);\ \mathit{mc}^{act} : \mathsf{ok}^{psv}}$$

Table C.4: Well-typedness of dynamic specification code mc_{sl}

Moreover, for a configuration $c_{sl} = (h, \mathsf{v}, (\mu, mc))$ of the specification language, we say that c_{sl} is *well-typed* regarding Γ and Δ, written

$$\Gamma;\Delta \vdash_{\mathsf{var}} c_{sl} : \mathsf{ok},$$

if

$$\Gamma;\Delta \vdash_{\mathsf{var}} \mathsf{v}\cdot\mu : \mathsf{ok} \quad \text{and if the judgment} \quad \Gamma;\Delta \vdash mc : \mathsf{ok}^{\gamma}$$

is derivable regarding the inference rules given in Table 3.2 and Table C.4.

While we have just seen that a configuration has to provide certain features, such that the method bodies of p can be executed properly, we still have to formulate the requirement that, contrary, p indeed provides method code that matches the expectations specified within the configuration specification. In particular, the code provided by p has to match the expectations in such a way that for each incoming call statement regarding method m within the configuration specification, we can find corresponding code in the method definition of m within p. This requirement is defined as follows.

Definition C.3.6 (Expectation supporting code)**:** Let mc_{sl} be activation record code regarding the specification language which is annotated with expectation ids. A program p with anticipation-based code structure *supports all expectations* of mc_{sl}, written

$$p \rhd mc_{sl},$$

if

- for each

  $$\big([i]\ (C\ x)?m(\overline{T}\ \overline{x})\texttt{.where}(e)\ \{\mathit{stmt}_{sl};\ \texttt{return}(e_r)\ \}\big)\ \in mc_{sl},$$

  there exist a corresponding conditional branch in the method definition of m in p such that

  $$\big(\mathsf{if}((\mathit{next} == i)\ \&\&\ (e))\ \{\ \mathit{stmt}_{pl};\ \mathit{retVal} = e_r\ \}\ \texttt{else}\ \mathit{stmt}'_{pl}\big)\ \in p.C.m$$

  with $\mathit{stmt}_{sl} \sim_{st} \mathit{stmt}_{pl}$.

- for each

  $$\big([i]\,\texttt{new}(C\ x)?C(\overline{T}\ \overline{x})\texttt{.where}(e)\ \{\mathit{stmt}_{sl};\ \texttt{return}\}\big)\ \in mc_{sl},$$

there exist a corresponding conditional branch in the constructor definition of C in p such that

$$\big(\mathsf{if}((next == i)\ \&\&\ (e))\ \{\ stmt_{pl}\}\ \texttt{else}\ stmt'_{pl}\big)\ \in p.C.m$$

with $stmt_{sl} \sim_{st} stmt_{pl}$.

Moreover, each expectation identifier that occurs within a conditional branch of a method or a constructor definition is unique.

Finally, we can define the relation R_t^p.

Definition C.3.7 (Testing bisimulation relation R_t^p)**:** Assume a program p with an anticipation-based code structure. Further, assume a type mapping Δ such that for all methods m of all classes C in p and for all Boolean expression e_1 to e_n of m according to Definition C.3.4 it is

$$\Gamma_g, \Gamma_{C.m}; \Delta \vdash e_k : \mathsf{Bool},$$

where $\Gamma_{C.m}$ represents the local type mapping due to the formal parameters and local variables of $C.m$ according to Rule T-MDEF in Table 2.2 and Γ_g is the local type mapping that results from p's global variables according to Rule T-PROG'.

We define a relation $R_t^b \subseteq Conf_{sl} \times Conf_{pl}$ over configurations of the specification language and of the programming language as follows. For all heap functions h and all global variable functions v the relation R_t^p exactly consists of the following pairs: It is

$$((h, \mathsf{v}, \mathsf{CS}_{sl}), (h, \mathsf{v}, \mathsf{CS}_{pl})) \in R_t$$

if, and only if,

1. regarding the call stacks it is

$$\mathsf{CS}_{sl} = (\mu, mc_{sl}) \quad \text{and} \quad (\mu, mc_{sl}) \sim_{CS} \mathsf{CS}_{pl},$$

2. the program p supports all expectations of mc_{sl}, i.e.,

$$p \rhd mc_{sl},$$

3. the specification configuration is well-typed regarding the local type mapping Γ_g and the global type mapping Δ of p, i.e.,

$$\Gamma_g; \Delta \vdash_{\mathsf{var}} (h, \mathsf{v}, (\mu, mc_{sl}) : \mathsf{ok}.$$

and

4. the specification configuration is anticipation-valid, i.e.,

$$h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc_{sl} : \mathsf{anticip}$$

Note, the heap and the global variables of related configurations are identical. Moreover, the call stack of the specification's configuration consists of a single activation record, only, and it must be related to the call stack of the program's configuration in terms of the relation $\sim_{CS}$.

Note further that, according to the operational semantics of the specification language, the call stack of a specification's configuration always consists of only one activation record. Hence, the corresponding equation, $\mathsf{CS}_{sl} = \mathsf{AR}_{sl}$ in Definition C.3.7 does not represent a real restriction.

Now, the following lemma will show that the relation R^p_t is a testing bisimulation as defined in 4.4.6. To understand the structure of the lemma's proof, recall that the code mc of a configuration's activation record is always either active, mc^{act}, or passive, mc^{psv}, code. In particular, it is always of the following form:

$$\begin{array}{lll} mc^{act} & ::= & s^{act} \mid s^{act};\ \texttt{!return}(e);\ mc^{psv} \\ mc^{psv} & ::= & s^{psv} \mid s^{psv};\ x = \texttt{?return}(T\ x).\texttt{where}(e);\ mc^{act} \end{array}$$

That is, the code of an activation record either consists of single statement (s^{act} or s^{psv}, respectively) or it consists of a statement followed by a return term and some more activation record code mc^{psv} or mc^{act}.

The proof of the lemma consists of a case analysis regarding the construction of the specification configurations of the relation R^p_t.

Lemma C.3.8: The binary relation R^p_t, defined in C.3.7, indeed represents a testing bisimulation as defined in 4.4.6.

Proof. Assume a program p with anticipation-based code structure. Further, assume a specification language configuration c_{sl} and a programming language specification c_{pl}, such that

$$(c_{sl}, c_{pl}) \in R^p_t. \tag{Ass}$$

The definition of R^p_t implies that there exist a heap function h, a global variable function v, as well as an activation record of the specification language $\mathsf{AR}_{sl} = (\mu, mc_{sl})$ and a call stack of the programming language CS_{pl} such that

$$c_{sl} = (h, \mathsf{v}, \mathsf{AR}_{sl}) \quad \text{and} \quad c_{pl} = (h, \mathsf{v}, \mathsf{CS}_{pl}) \quad \text{with } \mathsf{AR}_{sl} \sim_{CS} \mathsf{CS}_{pl}.$$

Similar to the proof of Lemma C.1.3, we make a *case analysis* regarding the construction of the code mc_{sl} of AR_{sl}. For each case we will prove that c_{pl} simulates c_{sl} ($\Rightarrow$) and additionally that c_{sl} simulates c_{pl} up to test faults ($\Leftarrow$). Specifically, we have to show *for each case* that the two configurations allow for similar computations steps where the resulting configurations, c'_{sl} and c'_{pl}, again meet the four requirements of Definition C.3.7. Two of the four requirements, however, can be shown generally without analyzing distinct cases. For, we have already shown in Lemma C.2.6 that anticipation validity is invariant concerning computation steps of the operational semantics. Moreover, it is obvious that, if p supports all expectations that are specified in c_{sl} then no computation step adds new expectations, so that p also supports all expectations specified in the new configuration c'_{sl}.

As for the following case analysis, we first consider the cases, where AR_{sl} contains active code mc^{act}. Afterwards, we consider all cases, where the code of AR_{sl} is passive, hence, an instance of mc^{psv}.

Case $\mathsf{AR}_{sl} = (\mathsf{v}_l{\cdot}\mu',\ s^{act};\ \texttt{!return}(e);\ mc^{psv})$ with $s^{act} \neq \epsilon$
Thus, the configurations c_{sl} is of the following form

$$c_{sl} = (h, \mathsf{v}, (\mathsf{v}_l{\cdot}\mu',\ s^{act};\ \texttt{!return}(e);\ mc^{psv})).$$

In particular, it is $\mu = \mathsf{v}_l{\cdot}\mu'$. So, according to Definition C.3.7 as well as Definition C.3.3, we know from (Ass) that

$$c_{pl} = (h, \mathsf{v}, (\check{\mathsf{v}}_l,\ stmt;\ retVal = e;\ \texttt{return}(retVal)) \circ \mathsf{CS}^{eb}),$$

such that

$$s^{act} \sim_{st} stmt \quad \text{and} \quad (\mu', mc^{psv}) \sim_{CS} \mathsf{CS}^{eb}.$$

We make a subcase analysis regarding the first active statement of s^{act}.

Subcase $s^{act} = x{=}e;\ s_1^{act}$
Then $s^{act} \sim_{st} stmt$ implies that

$$stmt = x{=}e;\ stmt_1 \quad \text{with } (*)\ s_1^{act} \sim_{st} stmt_1.$$

Direction $\Rightarrow$
According to the operational semantics of the specification language, c_{sl} may reduce to c'_{sl} only in terms of an internal computation step such that

$$c_{sl} \rightsquigarrow c'_{sl} = (h, \mathsf{v}', (\mathsf{v}_l{\cdot}\mu',\ s_1^{act};\ \texttt{!return}(e);\ mc^{psv}))$$

Note that the local variables did not change as (Ass) implies that x is not a local variable or parameter. Thus, similarly, we have

$$c_{pl} \rightsquigarrow c'_{pl} = (h, \mathsf{v}', (\check{\mathsf{v}}_l,\ stmt_1;\ retVal = e;\ \texttt{return}(retVal)) \circ \mathsf{CS}^{eb}).$$

So due to (Ass) and $(*)$ it is

$$(\mathsf{v}_l{\cdot}\mu',\ s_1^{act};\ \texttt{!return}(e);\ mc^{psv}) \sim_{CS} (\check{\mathsf{v}}_l,\ stmt_1;\ retVal = e;\ \texttt{return}(retVal)) \circ \mathsf{CS}^{eb}).$$

Again, the assumption (Ass) and Rule T-Seq of Table 2.2 imply that

$$\Gamma_g; \Delta \vdash_{\mathsf{var}} (h, \mathsf{v}', (\mathsf{v}_l{\cdot}\mu', s_1^{act};\ \texttt{!return}(e);\ mc^{psv})) : \mathsf{ok}.$$

Thus, according to Definition C.3.7 we get

$$(c'_{sl}, c'_{pl}) \in R_t.$$

$\boxed{\textbf{Direction}}$ $\Leftarrow$
The variable x must not be the extra variable $retVal$. Furthermore, c_{pl} can only deterministically reduce to the above mentioned c'_{pl}. Hence, this proof direction results in the same configuration pair

$$(c'_{sl}, c'_{pl}) \in R_t.$$

$\boxed{\textbf{Subcase}}$ $s^{act} = e_c!m(\overline{e})\{s^{psv};\ [i]\, x = ?\texttt{return}(T\ x').\texttt{where}(e')\};\ s_1^{act}$
In particular due to Definition C.3.1, the assumption (Ass) implies

$$c_{pl} = (h, \mathsf{v}, (\check{\mathsf{v}}_l, e_c.m(\overline{e});\ stmt_1;\ retVal = e;\ \texttt{return}(retVal)) \circ \mathsf{CS}^{eb}).$$

$\boxed{\textbf{Direction}}$ $\Rightarrow$
Configuration c_{sl} reduces to c'_{sl} due to an outgoing method call. Hence,

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

with

$$a = \nu(\Theta').\langle call\ o.m(\overline{v})\rangle! \quad \text{such that} \quad o = [\![e_c]\!]_h^{\mathsf{v},\mu} \quad \text{and} \quad \overline{v} = [\![\overline{e}]\!]_h^{\mathsf{v},\mu}.$$

and

$$c'_{sl} = (h, \mathsf{v}, (\mathsf{v}_\perp \cdot \mu,\ s^{psv};\ [i]\, x = ?\texttt{return}(Tx').\texttt{where}(e');\ s_1^{act};\ !\texttt{return}(e);\ mc^{psv})).$$

In the following, let us refer to the code of c'_{sl} by mc'_{sl}. Note that the new local variable function is the completely undefined variable function $\mathsf{v}_\perp$, since the code of c_{sl} is free of local variable declarations.

As for the programming language configuration c_{pl}, the topmost statement of the topmost activation record is the outgoing call $e_c.m(\overline{e})$ which likewise leads to a transition labeled with the same communication label a, such that

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

with

$$c'_{pl} = (h, \mathsf{v}, (\check{\mathsf{v}}_l, \texttt{rcv}\ x{:}T;\ stmt_1;\ retVal = e;\ \texttt{return}(retVal)) \circ \mathsf{CS}^{eb}).$$

In the following, let us refer to the code of c'_{pl} by mc'_{pl}. According to (Ass) and Definition C.3.1, it is

$$(\mathsf{v}_\perp \cdot \mu,\ mc'_{sl}) \sim_{st} ((\check{\mathsf{v}}_l, \texttt{rcv}\ x{:}T;\ stmt_1;\ retVal = e;\ \texttt{return}(retVal)) \circ \mathsf{CS}^{eb}).$$

Furthermore, Rule T-CallOut of Table 3.2 and Rule T-s^{psv}-RetI of Table C.4 imply that

$$\Gamma_g; \Delta \vdash_{\mathsf{var}} (h, \mathsf{v}, (\mathsf{v}_\perp \cdot \mu,\ mc'_{sl})) : \mathsf{ok}.$$

Hence, it is

$$(c'_{sl}, c'_{pl}) \in R_t.$$

$\boxed{\textbf{Direction}}$ $\Leftarrow$
Similar to the previous subcase, the configuration c_{pl} allows at most the same labeled transition to the configuration c'_{pl} that was introduced in the above proof regarding the other implication direction. This results in the same configuration pair such that, again,

$$(c'_{sl}, c'_{pl}) \in R_t.$$

The other subcases are similar.

$\boxed{\boxed{\textbf{Case}}}$ $\mathsf{AR}_{sl} = (\mathsf{v}_l{\cdot}\mu',\ !\mathtt{return}(e);\ mc^{psv})$
Referring to Definition C.3.3, we can derive from (Ass), that

$$c_{sl} = (h, \mathsf{v}, (\mathsf{v}_l{\cdot}\mu', !\mathtt{return}(e);\ mc^{psv}))$$

and, on the other hand, that

$$c_{pl} = (h, \mathsf{v}, (\check{\mathsf{v}}_l, retVal = e; \mathtt{return}(retVal)) \circ \mathsf{CS}^{eb})$$

or

$$c_{pl} = (h, \mathsf{v}, (\check{\mathsf{v}}_l, \mathtt{return}(retVal)) \circ \mathsf{CS}^{eb}),$$

where we additionally know in the latter case that $\check{\mathsf{v}}_l(retVal) = [\![e]\!]_h^{\mathsf{v},\mu}$. Moreover, we know that

$$(\mu', mc^{psv}) \sim_{CS} \mathsf{CS}^{eb}.$$

$\boxed{\textbf{Direction}}$ $\Rightarrow$
The only transition that may originate from c_{sl} is the one that is labeled with an outgoing return label a such that

$$a = \nu(\Theta').\langle return(v)\rangle! \quad \text{with } v = [\![e]\!]_h^{\mathsf{v},\mu}.$$

More specifically, due to Rule RetO of Table 3.3 we get

$$\Delta \vdash c_{sl} : \Theta \xrightarrow{a} \Delta \vdash c'_{sl} : \Theta' \quad \text{with } c'_{sl} = (h, \mathsf{v}, (\mu', mc^{psv})).$$

It is easy to see that processing the programming language configuration c_{pl} leads to the same outgoing communication step – with an intermediate internal computation step, if the case may be. In particular, in both cases, it is $\check{\mathsf{v}}_l(retVal) = [\![e]\!]_h^{\mathsf{v},\mathsf{v}_l\cdot\mu'}$ right before the outgoing return is processed. Therefore, it is

$$\Delta \vdash c_{pl} : \Theta \overset{a}{\Longrightarrow} \Delta \vdash c'_{pl} : \Theta' \quad \text{with } c'_{pl} = (h, \mathsf{v}, \mathsf{CS}^{eb}).$$

The assumption (Ass) immediately yields that

$$(\mu', mc^{psv}) \sim_{CS} \mathsf{CS}^{eb}.$$

Well-typedness of c'_{sl} results from Rule T-s^{act}-RetOut such that

$$\Gamma_g; \Delta \vdash_{\mathsf{var}} c'_{sl} : \mathsf{ok}.$$

So, all in all we can infer that

$$(c'_{sl}, c'_{pl}) \in R_t.$$

$\boxed{\textbf{Direction}} \Leftarrow$
Again, c_{pl} deterministically evolves to the configuration c'_{pl} of the previous proof direction.
$\boxed{\boxed{\textbf{Case}}}$ $\mathsf{AR}_{sl} = (\mathsf{v}_l,\ s^{act})$
The proof of this case is almost identical to the previous two proof cases. Specifically, we only have to skip the proof obligation that the trailing call stack CS^{eb} relates to the corresponding specification code, as no trailing call stack exists in this case.
$\boxed{\boxed{\textbf{Case}}}$ $\mathsf{AR}_{sl} = (\mu,\ s^{psv};\ [i]\, x = ?\mathtt{return}(T\, x').\mathtt{where}(e');\ mc^{act})$
Due to Definition C.3.3, it is $\mu = \mathsf{v}_\bot{\cdot}\mathsf{v}_l{\cdot}\mu'$ so that

$$c_{sl} = (h, \mathsf{v}, (\mathsf{v}_\bot{\cdot}\mathsf{v}_l{\cdot}\mu',\ s^{psv};\ [i]\, x = ?\mathtt{return}(T\, x').\mathtt{where}(e');\ mc^{act}).$$

Moreover the same definition leads to

$$\begin{aligned} &c_{pl} = (h, \mathsf{v}, (\check{\mathsf{v}}_l, \mathtt{rcv}\ x{:}T;\ \mathit{check}(i, e');\ mc) \circ \mathsf{CS}^{eb}) \quad \text{with} \\ &(\mathsf{v}_l, mc^{act}) \sim_{CS} (\check{\mathsf{v}}_l,\ mc) \circ \mathsf{CS}^{eb}. \end{aligned}$$

We consider some subcases regarding the structure of s^{psv}. However, this time we will not consider both implication directions for each subcase but only the simulation direction ($\Rightarrow$). We will prove the simulation-up-to-faults direction ($\Leftarrow$) for all subcases at the end.
$\boxed{\textbf{Subcase}}$ $s^{psv} = \mathsf{if}\ (e)\ \{s_1^{psv}\}\ \mathtt{else}\ \{s_2^{psv}\};\ s_3^{psv}$
Without loss of generality we can assume that $[\![e]\!]_h^{\mathsf{v},\mu} = \mathit{true}$ and thus

$$c_{sl} \rightsquigarrow c'_{sl} \quad \text{with} \quad c'_{sl} = (h, \mathsf{v}, (\mu,\ s_1^{psv};\ s_3^{psv};\ [i]\, x = ?\mathtt{return}(Tx').\mathtt{where}(e');\ mc^{act})).$$

However, again due to Definition C.3.3 it is

$$(\mu,\ s_1^{psv};\ s_3^{psv};\ [i]\, x = ?\mathtt{return}(T\, x').\mathtt{where}(e');\ mc^{act}) \sim_{CS} \mathsf{CS}_{pl}.$$

Due to Rule T-s^{act}-RetOut of Table C.4 and due to Rule T-Cond and Rule T-Seq of Table 3.2 we know that

$$\Gamma_g; \Delta \vdash_{\mathsf{var}} (\mu,\ s_1^{psv};\ s_3^{psv};\ [i]\, x = ?\mathtt{return}(T\, x').\mathtt{where}(e');\ mc^{act}) : \mathsf{ok}.$$

Thus, we get

$$(c'_{sl}, c_{pl}) \in R_t.$$

$\boxed{\textbf{Subcase}}$ $s^{psv} = [j]\,(C\,x)?m(\overline{T}\,\overline{x}).\mathtt{where}(e')\{\ s^{act};\ \mathtt{return}(e_r)\ \};\ s_3^{psv}$
In this case c_{sl} may only evolve due to an appropriate incoming method call label. That is,

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

with

$$c'_{sl} = (h, \mathsf{v}, (\mathsf{v}'_l{\cdot}\mu,\ s^{act};\ !\mathtt{return}(e_r)\ ;\ s_3^{psv};\ [i]\,x =?\mathtt{return}(Tx').\mathtt{where}(e');\ mc^{act}))$$

as well as

$$a = \nu(\Theta').\langle call\ o.m(\overline{v})\rangle? \quad \text{such that} \quad \Delta, \Delta', \Theta \vdash o, \overline{v} : C, \overline{T} \ \text{ and } \ [\![e']\!]_h^{\mathsf{v},\mathsf{v}'_l\cdot\mu}.$$

Let us refer to the code of c'_{sl} as mc'_{sl}. The assumption $h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc_{sl} : \mathsf{anticip}$ implies that

$$(*)\ [\![next]\!]_h^{\mathsf{v},\mu} = j$$

due to Lemma C.2.6. As for the configuration c_{pl}, the facts that p provides an anticipation-based code structure and, in particular, that $p \rhd mc_{sl}$, and finally that the program is generally input enabled, lead to

$$\Delta \vdash c_{pl} : \Theta \xrightarrow{a}_p \Delta' \vdash c'_{pl} : \Theta,$$

with

$$c'_{pl} = (h, \mathsf{v}, (\check{\mathsf{v}}'_l,\ stmt;\ \mathtt{return}(retVal)) \circ (\check{\mathsf{v}}_l,\ \mathtt{rcv}\ x{:}T;\ \ mc) \circ \mathsf{CS}^{eb}).$$

such that, due to $(*)$, it is $c'_{pl} \leadsto^* c''_{pl}$ with

$$c''_{pl} = (h, \mathsf{v}, (\check{\mathsf{v}}'_l,\ stmt_1;\ retVal = e_r;\ \mathtt{return}(retVal)) \circ (\check{\mathsf{v}}_l,\ \mathtt{rcv}\ x{:}T;\ mc) \circ \mathsf{CS}^{eb})$$

and with

$$s^{act} \sim_{st} stmt_1.$$

Let us refer to the code of the topmost activation record of c''_{pl} as mc''_{pl}. Then it is

$$(\mathsf{v}'_l{\cdot}\mu, mc'_{sl}) \sim_{st} (\check{\mathsf{v}}'_l, mc''_{pl}) \circ \mathsf{CS}^{eb}.$$

Due to Rule T-s^{psv}-RetI and Rule T-CallIn it is

$$\Gamma_g; \Delta \vdash_{\mathsf{var}} c'_{sl} : \mathsf{ok}$$

and finally we get

$$(c'_{sl}, c''_{pl}) \in R_t.$$

$\boxed{\textbf{Direction}}$ $\Leftarrow$
As mentioned above, the call stack CS_{pl} of the program configuration c_{pl} is externally blocked. Thus, it may only evolve due to an incoming call or due to an incoming return. That is, we can assume that

$$\Delta \vdash c_{pl} : \Theta \xrightarrow{a}_p \Delta' \vdash c'_{pl} : \Theta.$$

And regarding the communication label a we have to differentiate two subcases.
$\boxed{\textbf{Subcase}}$ $a = \nu(\Delta_n).\langle \mathit{call}\ o.m(\overline{v})\rangle?$
Due to the anticipation-based code structure of p, the configuration c'_{pl} is of the following form:

$$c'_{pl} = (h, \mathsf{v}, (\check{\mathsf{v}}_l, \mathit{stmt};\ \texttt{return}(\mathit{retVal})) \circ \mathsf{CS}_{pl}),$$

where *stmt* implements a case switch regarding expectation ids in form of a nesting of conditional statements as described in Definition C.3.4. Assume that

$$(*)\ \mathsf{v}(\mathit{next}) = j.$$

$\boxed{\textit{Subsubcase}}$ $\texttt{if}\ ((\mathit{next} == j)\&\&(e_j))\ \{\mathit{stmt}_j;\ \mathit{retVal} = e'_j\}\ \texttt{else}\ \{\mathit{stmt}'\}\ \in \mathit{stmt}$
Due to fact that p supports all expectations of the code of c_{sl}, i.e.,

$$p \rhd s^{psv};\ [i]\, x =?\texttt{return}(T\ x').\texttt{where}(e');\ mc^{act},$$

we can infer that $j \neq i$. Moreover, (Ass) implies that

$$h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc_{sl} : \mathsf{anticip}$$

so Lemma C.2.6 and $(*)$ yield that

$$\begin{aligned} &c_{sl} \rightsquigarrow^* c'_{sl} \quad \text{with} \\ &c'_{sl} = (h, \mathsf{v}, (\mu,\ [j]\, \mathit{stmt}_{in};\ s_1^{psv}; \\ &\qquad\qquad\qquad [i]\, x =?\texttt{return}(T\ x').\texttt{where}(e');\ mc^{act})). \end{aligned}$$

Again, since p supports all expectations of c_{sl}, it is indeed

$$\mathit{stmt}_{in} = (C\ x)?m(\overline{T}\ \overline{x}).\texttt{where}(e_j)\{\ s^{act};\ !\texttt{return}(e'_j)\ \}\ .$$

If $[\![e_j]\!]_h^{\mathsf{v},\mu_l\cdot\mu} = \mathit{false}$ then $\Delta \vdash c'_{sl} : \Delta \overset{a}{\not\to}$. But in this case also the corresponding conditional branch of m within p is evaluated to false such that the method reports a failure.

So let us assume that $[\![e_j]\!]_h^{\mathsf{v},\mu_l\cdot\mu} = \mathit{true}$. Then we get

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

with

$$c''_{sl} = (h, \mathsf{v}, (\mathsf{v}_l\cdot\mu, s^{act}; !\texttt{return}(e_j);\ s_1^{psv};\ ?\texttt{return}(T\ x').\texttt{where}(e');\ mc^{act})).$$

Let us refer to the activation record of c''_{sl} as AR''_{sl}. On the other hand, the program configuration c'_{pl} reduces to

$$c'_{pl} \leadsto^* c''_{pl} = (h, \mathsf{v}, (\check{\mathsf{v}}_l, stmt_j;\ retVal = e'_j;\ \mathtt{return}(retVal)) \circ \mathsf{CS}_{pl}),$$

where, yet again due to the expectation support, it is

$$(**)\ \ s^{act} \sim_{st} stmt_j.$$

Let us refer to the call stack of c''_{pl} as CS''_{pl}, then we get from (Ass) and from $(**)$ that

$$\mathsf{AR}''_{sl} \sim_{CS} \mathsf{CS}''_{pl}.$$

$\boxed{\textit{Subsubcase}}$ $\mathtt{if}\ ((next == j)\&\&(e_j))\ \{stmt_j;\ retVal = e'_j\}\ \mathtt{else}\ \{stmt'\} \not\in stmt$
That is, the method m does not provide a conditional branch regarding the communication identifier j. According to the structure of the method, this results in a failure report. Thus, we have to show that the specification configuration cannot realize an incoming call regarding a. Indeed, since $h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc_{sl} : \mathsf{anticip}$, we know from Lemma C.2.6 and from $(*)$ that

$$\Delta \vdash c_{sl} : \Theta \overset{a}{\not\to}.$$

$\boxed{\textbf{Subcase}}$ $a = \nu(\Delta_n).\langle return(v)\rangle?$
According to the operational semantics and due to the form of c_{pl} it is

$$\Delta, \Delta_n \vdash v{:}T$$

so that

$$\Delta \vdash c_{pl} : \Theta \xrightarrow{a} \Delta, \Delta_n \vdash c'_{pl} : \Theta$$

with $c'_{pl} = (h, \mathsf{v}', (\check{\mathsf{v}}_l,\ check(i, e');\ mc) \circ \mathsf{CS}^{eb})$. Since we assume that $check(i, e')$ tests whether $next = i$ and e' evaluates to true, we can differentiate two subsubcases.

$\boxed{\textit{Subsubcase}}$ $[\![next == i]\!]_h^{\mathsf{v},\check{\mathsf{v}}_l} \wedge [\![e']\!]_h^{\mathsf{v},\check{\mathsf{v}}_l} = true$
In this case we can assume that

$$c'_{pl} \leadsto^* c''_{pl} = (h, \mathsf{v}', (\check{\mathsf{v}}_l,\ mc) \circ \mathsf{CS}^{eb}),$$

but also we know from $h, \mathsf{v}, \mu \vdash_{\mathsf{ad}} mc_{sl} : \mathsf{anticip}$ that $s^{psv} = \epsilon$ and thus

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

with

$$c'_{sl} = (h, \mathsf{v}', (\mathsf{v}_l{\cdot}\mu', mc^{act})).$$

Finally, both,

$$(\mathsf{v}_l \cdot \mu', mc^{act}) \sim_{CS} (\check{\mathsf{v}}_l,\ mc) \circ \mathsf{CS}^{eb}$$

as well as

$$h, \mathsf{v}', \mathsf{v}_l \cdot \mu' \vdash_{\mathsf{var}} mc^{act} : \mathsf{ok}$$

immediately follow from (Ass).

$\boxed{\textit{Subsubcase}}$ $[\![next == i]\!]_h^{\mathsf{v},\check{\mathsf{v}}_l} \wedge [\![e']\!]_h^{\mathsf{v},\check{\mathsf{v}}_l} = \mathit{false}$
In this case, we assume that *check*(i, e') reports a failure. The specification configuration, however, does not accept such an incoming return label a, hence,

$$\Delta \vdash c_{sl} : \Theta \overset{a}{\not\to} .$$

$\boxed{\boxed{\textbf{Case}}}$ $\mathsf{AR}_{sl} = (\mathsf{v}_l,\ s^{psv})$
Similar to the s^{act} case, this s^{psv} case, again, represents a simplified version of the previous case, as we can replay its proofs while omitting the proof obligations regarding the trailing call stack CS^{eb} and, respectively, mc^{act}. □

In order to finally prove the correctness of the code generation algorithm, we have to show that the initial configurations of a specification s and the initial configuration of the correspondingly generated test program p represent a pair of the testing bisimulation relation R_t^p.

Lemma C.3.9 (Correctness of the test code generation)**:** Assume a well-typed specification s. Moreover, let $s' = \mathit{prep}(s)$ be the specification that results from preprocessing s as defined in Definition 4.1.4 and let p be the correspondingly generated program according to the algorithm described in Section 4.3. If the main statement of s' is an active statement then

$$(c_{init}(s'), c_{init}(p)) \in R_t^p.$$

Otherwise it is

$$(c_{init}(s'), \overline{c_{init}}(p)) \in R_t^p.$$

In particular, it is $R_t^p \neq \emptyset$.

Proof. Assume a well-typed configuration

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

Let $s' = \mathit{prep}(s)$. Then, according to Definition 4.1.4 we have

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

where stmt' results

1. from enriching *stmt* with anticipation code by means of the code processing functions prep_{in} and prep_{out} and

2. from "globalizing" all local variables within *stmt*, meaning that each variable declaration and formal parameter within *stmt* has a global counterpart in $\overline{x'}$ such that $stmt'$ is free of local variable declarations (apart from formal parameters). Moreover, all occurrences of local variables and parameters within *stmt* are replaced by the corresponding global counterpart.

It is easy to see that well-typedness of s implies well-typedness of s', hence, let us assume that $\Delta \vdash s' : \Theta$. Further let us assume that p with

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

is the test program generated from s' as described in Section 4.3. According to the code generation algorithm, the class definitions $\overline{impldecl}$ are generated by means of the code generation functions $code_{in}$ and $code_{out}$. From, the definitions of these functions, given in Table 4.5 and Table 4.6 as well as the auxiliary notation in Table 4.4 it immediately follows that p provides an anticipation-based structure. Moreover, the recursively descending application of $code_{in}$ and $code_{out}$ ensures that p supports all expectations of $stmt'$. It is

$$c_{init}(s') = (h_\perp, \mathsf{v}, (\mathsf{v}_\perp, stmt')),$$

where v maps each global variable of s' to its initial value. Well-typedness of s' implies that

$$\Gamma_g; \Delta \vdash_{\mathsf{var}} (h_\perp, \mathsf{v}, (\mathsf{v}_\perp, stmt')),$$

where Γ_g represents the local type mapping regarding the global variables (cf. Rule T-SPEC in Table 3.2). According to Definition C.3.7, it remains to show that the call stacks of the initial configurations of s and p are in relation regarding $\sim_{CS}$.

Case $stmt'$ is an active statement
In this case, consider

$$c_{init}(p) = (h_\perp, \mathsf{v}, (\mathsf{v}_\perp, stmt_{pl};\ \texttt{return}));$$

Since $stmt_{pl}$ results from applying $code_{out}$ to $stmt'$ we know from Lemma C.3.2 that

$$stmt' \sim_{st} stmt_{pl} \quad \text{hence} \quad (\mathsf{v}_\perp, stmt') \sim_{CS} (\mathsf{v}_\perp, stmt_{pl}).$$

Case $stmt'$ is a passive statement
In this case, consider

$$\overline{c_{init}}(p) = (h_\perp, \mathsf{v}, (\mathsf{v}_\perp, \epsilon));$$

Since $stmt'$ is an instance of s^{psv}, Definition C.3.3 yields

$$(\mathsf{v}_\perp, stmt') \sim_{CS} (\mathsf{v}_\perp, \epsilon).$$

□