Testing object Interactions

Grüner, A.

Chapter 7

Test specification language and code generation

As in the sequential setting, the underlying idea of our testing approach also in the multi-threaded setting is to provide a test specification language which allows to specify the *interface interactions* that may occur between the component under test and its environment. Regarding the sequential setting, the specification language's look-and-feel resembles that of the programming language *Japl* but also the specification language's semantics was geared towards *Japl*'s trace semantics. Describing a desired interaction trace, i.e., a *sequence* of communication labels, a *Japl* test specification, in particular, specifies the exact order of the entailed interface interactions. The consequence is that, in *Japl* we only need a single, sequentially composed, (main) specification statement which likewise stipulates an exact order due to its sequential construction.

In the multi-threaded setting, a trace of the semantics also represents a sequence of interactions. Due to the non-deterministic scheduling policy of the language, however, we cannot assure a certain sequence in general: if a program realizes a certain trace, then it also realizes different possible interleavings of the original trace. On account of this, we want to allow for specifying tests that are relaxed regarding the order of interactions carried out by different threads. Interactions that belong to the same thread, however, must again comply with a certain order. Therefore the idea is that the concurrent specification language shall allow to provide a specification statement *for each* thread that becomes active in the specification. To achieve this, we first have to identify the different situations in which a thread (identifier) may show up in a *CoJapl* program for the first time. There exist four different ways which are:

- internal thread creations due to instantiation of a thread class of the program

itself; for instance:

$$x = \texttt{spawn}\ \mathit{MyThread}(e, \ldots, e),$$

- outgoing spawn labels, that is, the program instantiates an external thread class; for instance:

$$a = \nu(\Delta', \Theta').\langle \mathit{spawn}\ n\ \mathit{of}\ C(\overline{v})\rangle!,$$

- incoming spawn labels, that is, the program's environment instantiates a thread class of the program, resulting in a communication label, like:

$$a = \nu(\Delta', \Theta').\langle \mathit{spawn}\ n\ \mathit{of}\ C(\overline{v})\rangle?,$$

- and incoming method or constructor call labels where the call is carried out by a new thread; for instance:

$$a = \nu(\Delta').n\langle \mathit{call}\ o.m(\overline{v})\rangle?, \quad \text{such that } n \in \Delta'.$$

For each of the above listed situations, a specification must provide a corresponding specification statement that determines the desired sequence of interface interaction carried out by the new thread. As in the single threaded case, we want to accomplish this by extending the programming language *CoJapl* with dedicated specification constructs.

Let us start with a spawn statement that results in an outgoing spawn label. That is, the specification instantiates a thread class C of the component under test. Following the style of the outgoing call statement of the sequential specification language, the spawn statement of the multi-threaded specification language resembles the original spawn statement but is equipped with an exclamation mark to indicate the cross-border communication.

$$x = \texttt{spawn!}C(e, \ldots, e).$$

A crucial difference between a method call specification statement (as well as a constructor call specification statement) on one hand and the spawn specification statement on the other hand is that the spawn statement is not split into two parts at the equal sign. For, the thread that carries out the spawn statement does not get blocked but always immediately returns such that the thread cannot realize any other actions in between the spawn and the corresponding return of the thread identifier. Since the spawn statement causes the creation of a new thread, the specification shall entail a description of the desired interface interactions realized by the new thread. To this end, we introduce the following *test thread construct* for specifying the interface behavior of a thread class C pertaining to the component under test:

$$\texttt{test thread}\ C(\overline{T}\,\overline{x})\{\ \mathit{stmt}\ \}.$$

According to our example, the above mentioned outgoing spawn statement results in a new thread of thread class C of the component under test and this new thread shall expose a behavior that conforms to the specification statement *stmt*. Note that *stmt* is parameterized regarding the spawn's parameters. Also note that the statement will be typed in a passive control context, as C is defined within the external component, hence, the thread becomes active in the external component, as well.

It has been mentioned, however, that a thread of an external thread class can also be created by the external component itself such that the specification does not know anything about the thread until it passes the interface for the very first time due to an incoming method or constructor call. In these cases the specification has to provide a desired behavior for the new thread, as well. Though, the corresponding specification statement cannot be parameterized regarding the spawn parameters, as it was not the specification that causes the thread spawning but the external component, hence, the corresponding actual parameters are not known to the specification. Therefore, we introduce a second test thread construct for specifying the behavior of thread classes of the component under test. In particular, it is almost identical to the aforementioned parameterized thread specification construct except that it does not provide any parameters. Therefore, the specification construct

$$\texttt{test thread}\ C\{\ \mathit{stmt}\ \}$$

means that, if a thread of class C enters the specification via a method or constructor call for the first time, then the interface behavior realized by this thread has to comply with the specification statement *stmt*.

Similar to the *CoJapl* programming language, a specification may not only spawn threads of externally defined thread classes but it also may create threads by means of internal thread creations, that is, the specification also supports the following spawn statement:

$$x = \texttt{spawn}(e, \ldots, e).$$

Note, however, in contrast to a *CoJapl* program, a specification may only use this statement in order to realize internal thread creations, since external thread creations are implemented by the above mentioned outgoing spawn statement.

A thread that comes to existence due to an internal thread creation also has to stick to a specified interface interaction sequence. Therefore, the specification language also provides a *mock thread construct* that stipulates a certain interface interaction due to a thread of a specification thread class C:

$$\texttt{mock thread}\ C(\overline{T}\,\overline{x})\{\ \mathit{stmt}\ \}.$$

Since a thread of a specification thread class C always starts in the specification, it consequently may pass the interface for the first time due to an outgoing communication, only. Hence, the specification statement *stmt* is typed in an active control context.

A thread of a specification thread class can come into existence either due to an internal spawn statement or due to an incoming spawn label. In both cases, the corresponding thread creation parameters are observable to the specification. Therefore, regarding specification thread classes, we do not need an additional mock thread construct that lacks the parameters.

Note, furthermore, that we need not to provide a specification statement for the expectation of *incoming* spawns. To understand the reason, consider the case that we do provide such a specification statement. More specifically, assume a specification of a thread n which entails the following fictitious incoming spawn statement:

$$x = \texttt{spawn}?C(\overline{T}\,\overline{x}).\texttt{where}(e).$$

The statement specifies that we expect the component under test to provoke an incoming spawn label via thread n resulting in a new thread. Let us assume, the name of the new thread is n'. Within the thread n itself, however, we cannot check if a spawn was executed. Also the new thread n' cannot verify that it was created by the specified spawn, as the originator of a spawn is unknown to the new thread, in general. Finally, due to the scheduling policy, the execution of a spawn statement and the resulting execution of the corresponding thread body are *decoupled* such that the start of the new thread does not allow to infer the point of time, when the spawn statement has been executed. For instance, even if the component under test executes, as specified by the above spawn expectation statement, the right spawn statement at the desired point of time, then the new thread n' may be scheduled much later. The conclusion is, neither can the specification observe the originator nor the point of time regarding an incoming spawn rendering it useless to introduce a corresponding specification statement.

Summarizing, the above mentioned mock thread construct is used for, both, internally spawned and externally spawned threads of specification thread classes. Now that we have discussed the new specification construct, the following section provides the syntax of the concurrent specification language, at large.

7.1 Syntax

The syntax of the test specification language for testing *CoJapl* components is given in terms of a grammar definition in Table 7.1. The language is basically an extension of the specification language, given in Table 3.1, by the interactions specification of thread classes. To this end, the language provides the new constructs that we have introduced in the previous section. In particular, the test thread constructs extend the declaration of the test unit classes while the mock thread construct completes the mock class declarations. We assume that the thread class names of all kinds of thread specification constructs are different. Note that, due to simplicity, this also means that the behavior of a thread class of the component under test may only be specified *either* by means of a parameterized test thread specification construct *or* by the non-parameterized test thread construct. Consequently, we assume that all instances of an external thread class may show up

s ::=	$\overline{cutdecl};\ \overline{mokdecl};\ \overline{T}\,\overline{x};\ \{\ stmt\ \}$	specification
$cutdecl$ ::=	`test class` C	test unit classes
	| `test thread` $C(\overline{T}\,\overline{x})\{\ stmt\ \}$	
	| `test thread` $C\{\ stmt\ \}$	
$mokdecl$ ::=	`mock class` $C\{C(T,\ldots,T);\ \overline{T\ m(T,\ldots,T)}\}$	mock classes
	| `mock thread` $C(\overline{T}\,\overline{x})\{\ stmt\ \}$	
$stmt$ ::=	$x = e \mid x =$ `new` $C \mid \varepsilon \mid stmt; stmt \mid \{\overline{T}\,\overline{x};\ stmt\}$	statements
	| `while` $(e)\ \{stmt\}$ | `if` $(e)\ \{stmt\}$ `else` $\{stmt\}$	
	| $x =$ `spawn` $C(e,\ldots,e)$	
	| $stmt_{in} \mid stmt_{out} \mid$ `case` $\{\ \overline{stmt_{in}; stmt}\ \}$	
$stmt_{in}$ ::=	$(C\,x)?m(\overline{T}\,\overline{x})$.`where`$(e)\ \{\overline{T}\,\overline{x};\ stmt;\ !$`return` $e\}$	incoming stmt
	| `new`$(C\,x)?C(\overline{T}\,\overline{x})$.`where`$(e)\{\overline{T}\,\overline{x};\ stmt;\ !$`return`$\}$	
$stmt_{out}$ ::=	$e!m(e,\ldots,e)\ \{\overline{T}\,\overline{x};\ stmt;\ ?$`return`$(x)$.`where`$(e)\ \}$	outgoing stmt
	| `new`$!C(e,\ldots,e)\{\overline{T}\,\overline{x};\ stmt;\ ?$`return`$(x)$.`where`$(e)\}$	
	| $x =$ `spawn`$!C(e,\ldots,e)$	
e ::=	$x \mid$ `null` | `op`$(e,\ldots,e) \mid$ `tid` | `tclass`	expressions

Table 7.1: Specification language for *CoJapl*: syntax

for the first time either due to an outgoing spawn or due to an incoming call.

Finally, the set of statements is extended by the internal and the outgoing spawn specification. Regarding the expressions, like in *CoJapl* we introduce the new expressions `tid` and `tclass` which allow a thread to determine its identifier and its class, respectively.

We conclude this section with two small examples which illustrate the usage of the specification language. The examples are given in Table 7.2. The first example on the left hand side of the table demonstrates the behavior specification of externally defined thread class, i.e., thread classes provided by the component under test. We assume that the component under test implements a thread that communicates with a (simplified) network service via its socket API (cf. [72], for instance). The used socket API, however, is wrapped in a *ServerSocket* class which is mocked by the specification. The thread under test is spawned by the main statement of the specification in Line 33. The thread specification is given in Lines 7 to 31. Specifically, the component (more precisely: its thread) is expected to create a *ServerSocket* object. Afterwards the component shall request the socket to listen to the network by means of an invocation of method *listen*. When the socket gets a connection request from the network it returns from the *listen* call. In this example, the method immediately return simulating a connection request. The component, in turn, accepts the connection by calling *accept* which is then followed by an undetermined number of *send* requests and by a final call of the *close* method.

The second example in Table 7.2 illustrates the specification of a mock thread class *StackTest* (Lines 2 to 26). It assumes an externally defined *Stack* class. More

specifically, the thread class tests if the *Stack* implementation is *thread-save*. That is, instances of the *Stack* class used via different threads must not interfere with each other resulting in an invalid stack structure. The thread class is parametrized in terms of three integer values which are pushed to and afterwards popped from a *Stack* object. Finally, the main specification statement spawns three threads of *StackTest* which, therefore, are executed concurrently.

Server socket example:

```
1  mock class ServerSocket{
2    ServerSocket ServerSocket();
3    bool listen();
4    ...
5  }
6
7  test thread ServSockThread() {
8    ServerSocket s;
9
10   new?(ServerSocket x)ServerSocket() {
11     s = x;
12     !return(s)
13   };
14   s?listen() {
15     !return(true)
16   };
17   s?accept() {
18     !return(true);
19   };
20   bool rcv = true;
21   while (rcv) {
22     case
23       s?send(Data d) {
24         !return(true)
25         }
26       s?close() {
27         rcv = false;
28         !return(true)
29         }
30   }
31 }
32 { thread x;
33   x = spawn!servSockThread()
34 }
```

Stack example:

```
1  test class Stack;
2  mock thread StackTest(int x1, x2, x3) {
3    Stack s;
4
5    new!Stack() {
6      ?return(s)
7    };
8    s!push(x1) {
9      ?return(1)
10   };
11   s!push(x2) {
12     ?return(2)
13   };
14   s!push(x3) {
15     ?return(3)
16   };
17   s!pop(){
18     ?return(x3)
19   }
20   s!pop(){
21     ?return(x2)
22   }
23   s!pop(){
24     ?return(x1)
25   }
26 }
27 { thread x;
28   x = spawn StackTest(1, 2, 3);
29   x = spawn StackTest(4, 5, 6);
30   x = spawn StackTest(7, 8, 9);
31 }
```

Table 7.2: *CoJapl* example specifications

7.2 Static semantics

The type system for the concurrent test specification language is given in Table 7.3. It extends the type system for the sequential language, given in Table 3.2, by new rules regarding the newly introduced constructs. Moreover, Rule T-SPEC requires a simple adaption, as the mock thread declarations have to be type-checked, while the mock class declarations of the sequential settings were only used to extract the type information.

Rule T-TESTT$_{Spwn}$ deals with the test thread specification of threads spawned by means of an outgoing spawn label. Thus, the corresponding thread class C has to be included in the assumption context. Moreover, its type must comply with the thread specification. Finally, the body statement *stmt* of the thread specification has to be well-typed regarding a type context that is enriched by the thread creation parameters. Specifically, the statement has to be a passive statement, since the thread starts within the component under test. Similarly, the Rule T-TESTT$_{Call}$ deals with the specification construct of an external thread that shows up in the specification due to an incoming call. Therefore, the specification construct is not parameterized by the thread creation parameters, so we can omit the type check of the previous rule. Yet, also here, the name C must be typed as an externally defined thread class. Likewise, the specification statement must be passive.

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

$$[\text{T-TESTT}_{Spwn}]\ \frac{\Delta \vdash C : \overline{T} \quad \Gamma' = \Gamma, \overline{x}{:}\overline{T} \quad \Gamma';\Delta;\Theta \vdash \mathit{stmt} : \mathsf{ok}^{psv}}{\Gamma;\Delta;\Theta \vdash \texttt{test thread}\ C(\overline{T}\,\overline{x})\{\ \mathit{stmt}\ \} : \mathsf{ok}}$$

$$[\text{T-TESTT}_{Call}]\ \frac{\Delta \vdash C : \overline{T} \quad \Gamma';\Delta;\Theta \vdash \mathit{stmt} : \mathsf{ok}^{psv}}{\Gamma;\Delta;\Theta \vdash \texttt{test thread}\ C\{\ \mathit{stmt}\ \} : \mathsf{ok}}$$

$$[\text{T-MOCKT}]\ \frac{\Gamma' = \Gamma, \overline{x}{:}\overline{T} \quad \Gamma';\Delta;\Theta \vdash \mathit{stmt} : \mathsf{ok}^{act}}{\Gamma;\Delta;\Theta \vdash \texttt{mock thread}\ C(\overline{T}\,\overline{x})\{\ \mathit{stmt}\ \} : \mathsf{ok}}$$

$$[\text{T-SPAWN}_i]\ \frac{\Gamma;\Delta,\Theta \vdash x{:}\texttt{thread} \quad \Theta \vdash C{:}\overline{T} \quad \Gamma;\Delta,\Theta \vdash \overline{e}{:}\overline{T}}{\Gamma;\Delta;\Theta \vdash x = \texttt{spawn}\ C(\overline{e}) : \mathsf{ok}^{act}}$$

$$[\text{T-SPAWN}_{out}]\ \frac{\Gamma;\Delta,\Theta \vdash x{:}\texttt{thread} \quad \Delta \vdash C{:}\overline{T} \quad \Gamma;\Delta,\Theta \vdash \overline{e}{:}\overline{T}}{\Gamma;\Delta;\Theta \vdash x = \texttt{spawn!}C(\overline{e}) : \mathsf{ok}^{act}}$$

Table 7.3: Specification language for *CoJapl*: type system (stmts)

The mock thread specification represents the dual of the test thread specification. Thus, its typing rule T-MockT is almost identical to Rule T-TestT$_{spwn}$ but only its statement is type-checked in an active control context.

Finally, the two new spawn statements are type-checked by Rule Spawn$_i$ and Rule Spawn$_{out}$, respectively. In both cases, the variable x has to be a thread variable and the class name C must be appropriately typed as a thread class. Regarding the internal spawn statement, however, the thread class must be provided by the commitment context Θ while the outgoing spawn statement is only well-typed if the thread class can be found in the assumption context Δ. Note that both statements are only well-typed in an active control context.

7.3 Operational semantics

Again, the internal steps of the operational semantics are identical to the rules of the concurrent programming language *CoJapl*, hence, we do not repeat them again. Regarding the external steps, we adapt the rules of the sequential specification language by extending it with threads. This is done, as explained above, by exchanging the call stack of the configurations with a thread configuration mapping. Therefore, we also omit most of the rules inherited from the sequential setting. We add new rules for the new thread-related specification constructs. As with *CoJapl*, we have to differentiate incoming calls via new threads from rules regarding re-entrant threads. The new rules are shown in Table 7.4.

An incoming spawn causes the extension of the thread configuration mapping **tc**, where the new call stack is initialized with the specification statement of the corresponding mock thread specification. To this end, we redefine the code extracting function *cbody* such that it extracts the body statement from mock and test thread specifications. We omit the straightforward redefinition of *cbody*.

Similarly, an outgoing spawn causes the extension of **tc**, where the call stack is initialized with the specification statement of the corresponding test thread specification. Additionally, the call stack that implements the outgoing spawn statement is reduced and the global and local variables are updated with the new thread identifier.

Regarding incoming method and constructor calls, we have to provide two rules each, as explained above. Rule CalliI deals with incoming method calls realized by a thread n that is known to the specification, already. In particular, there exist a thread configuration for n in **tc** already, whose call stack specifies the expectation of this incoming call. Furthermore, the where-clause e' of the expectation evaluates to true, so the call stack is reduced and the thread configuration mapping is correspondingly updated.

Rule CalliI$_{nt}$, in contrast, deals with incoming calls that are realized by means of a new thread. Thus, the thread configuration mapping does not provide a corresponding mapping. However, the rules requires that a test thread specification regarding this thread is provided, such that the thread specification's first expectation statement matches with the incoming call. In this case, the thread configuration mapping is extended by a new thread configuration for the new

$$[\textsc{SpawnI}]\ \frac{\begin{array}{c} a = \nu(\Delta', \Theta').\langle \mathit{spawn}\ n\ \mathit{of}\ C(\overline{v})\rangle?\quad \Delta \vdash a : \Theta \\ \mathbf{tc}' = \mathbf{tc}[n \mapsto (C, (\mathsf{v}_l, \mathit{tbody}(C))]\end{array}}{\Delta \vdash (h, \mathsf{v}, \mathbf{tc}) : \Theta \xrightarrow{a} \Delta, \Delta' \vdash (h, \mathsf{v}, \mathbf{tc}') : \Theta, \Theta'}$$

where $\overline{T}\,\overline{x} = \mathit{tparams}(C)$ and $\mathsf{v}_l = \{\mathtt{tid} \mapsto n, \mathtt{tclass} \mapsto C, \overline{x} \mapsto \overline{v}\}$

$$[\textsc{SpawnO}]\ \frac{\begin{array}{c} a = \nu(\Theta', \Delta').\langle \mathit{spawn}\ n\ \mathit{of}\ C(\overline{v})\rangle! \\ \mathbf{tc}(n').cs = (\mu, x = \mathtt{spawn}!C(\overline{e}); mc) \circ \mathsf{CS} \\ \mathbf{tc}' = \mathbf{tc}[n' \mapsto (\mu', mc) \circ \mathsf{CS}][n \mapsto (C, (\mathsf{v}_l, \mathit{tbody}(C)))]\end{array}}{\Delta, \Delta' \vdash (h, \mathsf{v}, \mathbf{tc}) : \Theta \xrightarrow{a} \Delta \vdash (h, \mathsf{v}', \mathbf{tc}') : \Theta, \Theta'}$$

where $\overline{v} = [\![\overline{e}]\!]_h^{\mathsf{v},\mu}$, $n \in N \setminus \mathit{dom}(\mathbf{tc})$, $(\mathsf{v}', \mu') = \mathit{vupd}(\mathsf{v}, \mu, x \mapsto n)$, $\Delta' = (n{:}C)$, $\Theta' = \mathtt{new}(h, \overline{v}, \Theta)$, and $\mathsf{v}_l = \{\mathtt{tid} \mapsto n, \mathtt{tclass} \mapsto C, \overline{x} \mapsto \overline{v}\}$

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

where $C = \Theta(o)$, $\Delta, \Theta \vdash n : C_T$, and $\mathsf{v}_l = \{\mathtt{tid} \mapsto n, \mathtt{tclass} \mapsto C_T, \mathtt{this} \mapsto o, \overline{x} \mapsto \overline{v}, \overline{x'} \mapsto \mathit{ival}(\overline{T'})\}$

$$[\textsc{CallI}_{nt}]\ \frac{\begin{array}{c} a = \nu(\Delta').n\langle \mathit{call}\ o.m(\overline{v})\rangle?\quad \Delta' \vdash n : C_T \\ \Delta \vdash a : \Theta\quad \mathit{tbody}(C_T) = s^{psv};\ s_1^{psv} \\ s^{psv} = (C\ x)?m(\overline{T}\ \overline{x}).\mathtt{where}(e')\{\overline{T_l}\ \overline{x_l};\ s^{act};\ !\mathtt{return}\ e\} \\ [\![e']\!]_h^{\mathsf{v},\mathsf{v}_l}\quad \mathbf{tc}' = \mathbf{tc}[\ n \mapsto (\mathsf{v}_l, s^{act};\ !\mathtt{return}\ e;\ s_1^{psv})]\end{array}}{\Delta \vdash (h, \mathsf{v}, \mathbf{tc}) : \Theta \xrightarrow{a} \Delta, \Delta' \vdash (h, \mathsf{v}, \mathbf{tc}') : \Theta}$$

where $C = \Theta(o)$, and $\mathsf{v}_l = \{\mathtt{tid} \mapsto n, \mathtt{tclass} \mapsto C_T, \mathtt{this} \mapsto o, \overline{x} \mapsto \overline{v}, \overline{x'} \mapsto \mathit{ival}(\overline{T'})\}$

Table 7.4: Specification language for *CoJapl*: operational semantics (external)

thread where the call stack consists of the thread's specification statement. We skip the rules for incoming constructor call, as they are very similar to the rules for incoming method calls.

7.4 Test code generation

In this section we want to sketch a possible extension of the sequential code generation algorithm achieving a code generation algorithm for the multi-threaded setting.

Recall, that a central idea of the sequential test code generation was the *anticipation* of the next incoming communication. Specifically, in the sequential setting we annotated each incoming communication term of the specification with an expectation identifier. On the other hand, we added a global variable *next* which provided the label of the next upcoming incoming communication term. This way, we could distribute the (translated) code of the specification over several method definitions without loosing track of the stipulated sequential order of the specified interface interactions.

As for the multi-threaded setting, we will embark exactly on the same strategy, though the sequential order and the corresponding anticipation mechanism will be carried out for each thread, only. In particular, all specification statements of each mock thread and test thread specification are equipped with a unique expectation identifier annotation as well as with corresponding *next* update statements, so that, for instance, as a first approach, a mock thread specification

$$\texttt{mock thread}\ C(\overline{T}\,\overline{x})\{\ s^{act}\ \}$$

is preprocessed according to the preprocessing step as described in Section 4.1 resulting in a thread specification

$$\texttt{mock thread}\ C(\overline{T}\,\overline{x})\{\ prep_{out}(s^{psv})\ \},$$

where s^{psv}_{pp} results from applying $prep_{out}$ on s^{psv}. It is crucial, that each thread uses its own *next* variable since the specification stipulates the sequential order of interactions only per thread. In this context, a little complication arise from the fact that threads can be spawned dynamically. For, it is not sufficient to declare a static number of global *next* variables but instead we have to implement the *next* variables by means of a globally accessible dynamic list which maps thread identifiers to the corresponding next expectation identifier. We abstract the details regarding the list implementation away such that in the following we refer to the globally accessible list in terms of an array. That is, we assume that the preprocessed specification provides a dynamic list *next* where the expression *next*[*n*] yields the next expectation identifier for the thread with thread identifier n (if the list defines a value for n, at all).

As for the method code generation, the transition from the sequential to the multi-threaded setting is rather straightforward. Concerning the methods' case switches (cf. Section 4.2), we merely have to replace the *next* expression by a *next*[`tid`] expression. Thus, compared to Table 4.7 the case switch consists of conditional statements of the following form:

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

However, additionally we have to consider the case that a thread enters the test program for the first time via an incoming method or constructor call. In this case, *next*[`tid`] is not defined. Hence, we first have to check if the thread is new and, if so, we have to determine the matching thread specification for this thread. Note that only threads of externally defined thread classes may show up at the interface for the first time in terms of an incoming method or constructor call. Therefore, we can assume that the new thread is instantiated from an externally defined thread class and, correspondingly, only test thread specifications come into question for this matching procedure.

If a matching thread specification is found, then the *next* list is extended by `tid` such that $next[\texttt{tid}]$ is initialized with the first expectation identifier of the matching thread specification. For a better understanding, consider an example specification which includes a test thread specification regarding thread class C_T that starts with an incoming method call expectation of method m of class C, i.e.,

`test thread` $C_T\{\ [i](C\ x)?m(T\ y).\texttt{where}(e)\ \{\ \ldots$.

Moreover, assume that indeed an incoming call of method m of an instance of C via thread n has occurred, where n is new to the specification. In particular, $next[n]$ is not defined. Then method m has to set $next[n]$ to the expectation identifier i. Specifically, the above mentioned case switch in the body of method m has to be proceeded by the following code:

```
if (tid ∉ next) {
  if (tclass == C_T) {
    next[tid] = i
  } else {
    fail
  }
}
```

Thus, when `tid` is not in the *next* list, hence, when `tid` shows up for the first time, then method m checks the class type of the new thread by means of `tclass`. If the calling class is C_T then the *next* list is extended by `tid` that is mapped to the expectation identifier i. As we put this conditional statement at the very beginning of the method body, the above mentioned case switch can be executed subsequently.

So far, we have ignored two further problems that arise from the dynamic thread creation. First, we cannot resolve the local variable declaration problem with variable globalization, anymore (cf. Section 4.1.2). To understand this, consider the following test thread specification:

```
test thread C_T{
  [i](C x)?m1(T y).where(e) {
    o!m2() {
      (C x)?m1(T z).where(y = z) {
    ... },
```

The above specification consists of two nested incoming calls of $m1$ where the inner call's where-clause uses the parameter y of the outer call. As several instances of thread C_T may be created during the execution it is not sufficient to provide a (single) global pendant for y as we have done it in the sequential setting. Instead, for each thread we have to provide a corresponding set of global variables. Thus, similar to the solution for the global *next* variable, for each local variable we have to implement a dynamic list of globally accessible variables. Then, regarding the nested calls example given above, the second incoming call specification of $m1$ may access y by $y[\texttt{tid}]$.

enter:

```
1  access = false;
2  while (!access) {
3    while(accID != 0) { };
4    accID = accID + tid;
5    if (addID == tid)
6      { access = true }
7    else
8      { accID = accID − tid }
9  }
```

exit:

```
1  access = false;
2  accID = accID − tid;
```

Table 7.5: *CoJapl* code generation: mutual exclusion

The second problem is due to the fact that a globally accessible list implementation must only allow a *mutually exclusive* writing access to the list, in order to avoid inconsistency. Therefore, in the following we provide a simple mutual exclusion algorithm for *CoJapl*. In particular, we assume that writing accesses to global lists are only realized within *critical sections*. Table 7.5 sketches entry and exit code to be executed by a thread whenever it wants to enter and, respectively, exit a critical section. The only assumption for this algorithm concerning the *CoJapl* language is that we consider thread identifiers to be represented by *integers* which can be added and subtracted. Each thread has a local variable *access* and additionally all threads share a global variable *accID*. The local variable *access* is used by a thread to indicate that is has access to the critical section. The global variable *addID* stores thread identifiers of competing threads. Let us have a closer look at the entry code. After initializing *access* to false, we enter the while-loop at Line 2. After that, we have to busy-wait for *accID* to become 0. A value of 0 indicates that no thread is in the critical section and that currently no thread has requested entrance to the critical section. A thread requests for entrance to the critical section by incrementing *accID* with its own thread it. If then afterwards *accID* indeed stores the thread identifier of the thread, then the thread is allowed to enter the section. Since other threads may have incremented the variable concurrently as well, however, *accID* may be unequal to the thread identifier. In this case, all competing threads have to decrement *accID* by their thread identifier again. Due to the fact that an assignment represents an *atomic computation step* in our language, there exist at most one thread which may find *accID* to store exactly its own thread identifier. Therefore, mutual exclusion is granted. When leaving the critical section, the thread again subtracts its identifier from *accID*.