Asynchronous Programming in the Abstract Behavioural Specification Language

Azadbakht, K.

Chapter 6

Deadlock Detection for Actor-Based Coroutines

6.1 Introduction

Actors [2][47] provide an event-driven concurrency model for the analysis and construction of distributed, large-scale parallel systems. In actor-based modeling languages, like Rebeca [69], Creol [51], and ABS [48], the events are generated by *asynchronous calls* to methods provided by the actors. The resulting integration with object-orientation allows for new object-oriented models of concurrency, better suited for the analysis and construction of distributed systems than the standard model of *multi-threading* in languages like Java.

The new object-oriented models of concurrency arise from the combination of different synchronization mechanisms. By design, the basic *run-to-completion* mode of execution of asynchronously called methods as for example provided by the language Rebeca does not provide any synchronization between actors. Consequently, the resulting concurrent systems of actors do not give rise to undesirable consequences of synchronization like *deadlock*. The languages Creol and ABS extend the basic model with synchronization on the values returned by a method. So-called *futures* [27] provide a general mechanism for actors to synchronize on return values. Creol and ABS further integrate a model of execution of methods based on and inspired by *coroutines*, attributed by D. Knuth to M. Conway [24]. This model allows for controlled suspension and resumption of the executing method invocation and so-called cooperative scheduling of another method invocation of the actor.

Both the synchronization mechanisms of futures and coroutines may give rise to deadlock. Futures may give rise to *global* deadlock in a system of actors. Such a global deadlock consists of a circular dependency between different method invocations of possibly different actors which are suspended on the generation of the return value. On the other hand, coroutines may give rise to a *local* deadlock which occurs when all method invocations of a single actor are suspended on a Boolean

condition. In this chapter we provide the formal foundations of a novel method for the analysis of such local deadlocks.

To the best of our knowledge, our work provides a first method for deciding local deadlocks in actor-based languages with coroutines. The method itself is based on a new technique for predicate abstraction of actor-based programs with coroutines, which aims at the construction of a well-structured transition system. In contrast, the usual techniques of predicate abstraction [17] aim at the construction of a *finite* abstraction, which allows model checking of properties in temporal logic. In [29], a restricted class of actor-based programs is modeled as a well-structured transition system. This class does not support coroutines and actors do not have a global state specifying the values of the global variables.

Methods that utilize different techniques aiming at detection of global deadlocks in various actor settings include the following. The work in [53] uses ownership to organize CoJava active objects into hierarchies in order to prevent circular relationships where two or more active objects wait indefinitely for one another. Also data-races and data-based deadlocks are avoided in CoJava by the type system that prevents threads from sharing mutable data. In [26], a sound technique is proposed that translates a system of asynchronously communicating active objects into a Petri net and applies Petri net reachability analysis for deadlock detection. The work that is introduced in [38] and extended in [45] defines a technique for analyzing deadlocks of stateful active objects that is based on behavioural type systems. The context is the actor model with wait-by-necessity synchronizations where futures are not given an explicit "Future" type. Also, a framework is proposed in [52] to statically verify communication correctness in a concurrency model using futures, with the aim that the type system ensures that interactions among objects are deadlock-free.

A deadlock detection framework for ABS is proposed in [39] which mainly focuses on deadlocks regarding future variables, i.e., await and get operations on futures. It also proposes a naive annotation-based approach for detection of local deadlocks (await on Boolean guards), namely, letting programmers annotate the statement with the dependencies it creates. However, a comprehensive approach to investigate local deadlocks is not addressed. Our approach, and corresponding structure of the chapter, consists of the following. First, we introduce the basic programming concepts of asynchronous method calls, futures and coroutines in Section 6.2. In Section 6.3 we introduce a new operational semantics for the description of the local behavior of a single actor. The only external dependencies stem from method calls generated by other actors and the basic operations on futures corresponding to calls of methods of other actors. Both kinds of external dependencies are modeled by *non-determinism*. Method calls generated by other actors are modeled by the non-deterministic scheduling of method invocations. The basic operations on futures are modeled by the corresponding non-deterministic evaluation of the availability of the return value and random generation of the return value itself. Next, we introduce in Section 6.4 a *predicate abstraction* [17, 40] of the value assignments to

the global variables ("fields") of an actor as well as the local variables of the method invocations. The resulting abstraction still gives rise to an *infinite* transition system because of the generation of *self*-calls, that is, calls of methods of the actor by the actor itself, and the corresponding generation of "fresh" names of the local variables.

Our main contribution consists of the following technical results.

- a proof of the *correctness* of the predicate abstraction, in Section 6.5, and
- *decidability* of checking for the occurrence of a local deadlock in the abstract transition system in Section 6.7.

Correctness of the predicate abstraction is established by a *simulation* relation between the concrete and the abstract transition system. Decidability is established by showing that the abstract system is a so-called well-structured transition system, cf. [36]. Since the concrete operational semantics of the local behavior of a single actor is an *over-approximation* of the local behavior in the context of an arbitrary system of actors, these technical results together comprise a general method for proving *absence* of local deadlock of an actor. A short discussion follow-up in Section 6.8 concludes the chapter.

6.2 The Programming Language

In this section we present, in the context of a class-based language (with a subset of ABS features), the basic statements which describe asynchronous method invocation and cooperative scheduling.

A class introduces its global variables, also referred to as "fields", and methods. We use $x, y, z, \ldots$ to denote both the fields of a class and the local variables of the methods (including the formal parameters). Method bodies are defined as sequential control structures, including the usual conditional and iteration constructs, over the basic statements listed below.

Dynamic instantiation For x a so-called future variable or a class variable of type C, for some class name C, the assignment

$$x = \texttt{new}$$

creates a new future or a unique reference to a new instance of class C.

Side effect-free assignment In the assignment

$$x = e$$

the expression e denotes a side effect-free expression. The evaluation of such an expression does not affect the values of any global or local variable and also does

not affect the status of the executing process. We do not detail the syntactical structure of side effect-free expressions.

Asynchronous method invocation A method is called asynchronously by an assignment of the form

$$x = e_0!m(e_1, \ldots, e_n)$$

Here, x is a future variable which is used as a unique reference to the return value of the invocation of method m with actual parameters $e_1, \ldots, e_n$. The called actor is denoted by the expression e_0. Without loss of generality we restrict the actual parameters and the expression e_0 to side effect-free expressions. Since e_0 denotes an actor, this implies that e_0 is a global or local variable.

The get operation The execution of an assignment

$$x = y.\texttt{get}$$

blocks till the future variable y holds the value that is returned by its corresponding method invocation.

Awaiting a future The statement

$$\texttt{await}\ x?$$

releases control and schedules another process in case the future variable x does not yet hold a value, that is to be returned by its corresponding method invocation. Otherwise, it proceeds with the execution of the remaining statements of the executing method invocation.

Awaiting a Boolean condition Similarly, the statement

$$\texttt{await}\ e$$

where e denotes a side effect-free Boolean condition, releases control and schedules another process in case the Boolean condition is false. Otherwise, it proceeds with the execution of the remaining statements of the executing method invocation.

We describe the possible deadlock behavior of a system of dynamically generated actors in terms of *processes*, where a process is a method invocation. A process is either *active* (executing), *blocked* on a get operation, or *suspended* by a future or Boolean condition. At run-time, an actor consists of an active process and a set of suspended processes (when the active method invocation blocks on a get operation it blocks the entire actor). Actors execute their active processes in parallel and only interact via asynchronous method calls and futures. When an active process

awaits a future or Boolean condition, the actor can cooperatively schedule another process instead. A *global* deadlock involves a circular dependency between processes which are awaiting a future. On the other hand, a *local* deadlock appears when all the processes of an actor are awaiting a Boolean condition to become true. In the following sections we present a method for showing if an initial set of processes of an individual actor does *not* give rise to a local deadlock.

6.3 The Concrete System

In order to formally define local deadlock we introduce a formal operational semantics of a single actor. Throughout this chapter we assume a definition of a class C to be given. A typical element of its set of methods is denoted by m. We assume the definition of a class C to consist of the usual declarations of global variables and method definitions. Let $Var(C)$ denote all the global and local variables declared in C. Without loss of generality we assume that there are no name clashes between the global and local variables appearing in C, and no name clashes between the local variables of different methods. To resolve in the semantics name clashes of the local variables of the different invocations of a method, we assume a given infinite set Var such that $Var(C) \subseteq Var$. The set $Var \setminus Var(C)$ is used to generate "fresh" local variables. Further, for each method m, we introduce an infinite set $\Sigma(m)$ of renamings σ such that for every local variable x of m, $\sigma(x)$ is a fresh variable in Var, i.e. not appearing in $Var(C)$. We assume that any two distinct $\sigma, \sigma' \in \bigcup_m \Sigma(m)$ are *disjoint* (Here m ranges over the method names introduced by class C.) Renamings σ and σ' are disjoint if their ranges are disjoint. Note that by the above assumption the domains of renamings of *different* methods are also disjoint. By auxiliary function $fresh(\sigma')$ we check that the renaming $\sigma' \in \Sigma(m)$ is different from all the existing renamings in Q.

A process p arising from an invocation of a method m is described formally as a pair (σ, S), where $\sigma \in \Sigma(m)$ and S is the sequence of remaining statements to be executed, also known as *continuation*. An actor configuration then is a triple (Γ, p, Q), where Γ is an assignment of values to the variables in Var, p denotes the active process, and Q denotes a set of suspended processes. A configuration is *consistent* if for every renaming σ there exists at most one statement S such that $(\sigma, S) \in \{p\} \cup Q$.

A computation step of a single actor is formalized by a transition relation between consistent actor configurations. A structural operational semantics for the derivation of such transitions is given in Table 6.1. Here, we assume a given set Val of values of built-in data types (like Integer and Boolean), and an infinite set R of references or "pointers". Further, we assume a global variable *refs* such that $\Gamma(refs) \subseteq R$ records locally stored references.

We proceed with the explanation of the rules of Table 6.1. The rule <ASSIGN>

$$\frac{}{(\Gamma, (\sigma, x = e; S), Q) \to (\Gamma[x\sigma = \Gamma(e\sigma)], (\sigma, S), Q)}\ \texttt{<ASSIGN>}$$

$$\frac{r \in R \backslash \Gamma(\mathit{refs})}{(\Gamma, (\sigma, x = \texttt{new}; S), Q) \to (\Gamma[\mathit{refs} = \Gamma[\mathit{refs}] \cup \{r\}], (\sigma, x = r; S), Q)}\ \texttt{<NEW>}$$

$$\frac{v \in \mathit{Val}}{(\Gamma, (\sigma, x = \ y.\texttt{get}; S), Q) \to (\Gamma[x\sigma = v], (\sigma, S), Q)}\ \texttt{<GET-VALUE>}$$

$$\frac{r \in R}{(\Gamma, (\sigma, x = \ y.\texttt{get}; S), Q) \to (\Gamma[\mathit{refs} = \Gamma(\mathit{refs}) \cup \{r\}], (\sigma, x = r; S), Q)}\ \texttt{<GET-REF>}$$

$$\frac{\Gamma(y\sigma) \neq \Gamma(\texttt{this})}{(\Gamma, (\sigma, x = y!m(\bar{e}); S), Q) \to (\Gamma, (\sigma, x = \texttt{new}; S), Q)}\ \texttt{<REMOTE-CALL >}$$

$$\frac{\Gamma(y\sigma) = \Gamma(\texttt{this}) \quad \mathit{fresh}(\sigma')}{(\Gamma, (\sigma, x = y!m(\bar{e}); S), Q) \to (\Gamma[\bar{z}\sigma' = \Gamma(\bar{e}\sigma)], (\sigma, x = \texttt{new}; S), Q \cup \{(\sigma', S')\})}\ \texttt{<LOCAL-CALL>}$$

$$\frac{\Gamma(e\sigma) = \mathit{true}}{(\Gamma, (\sigma, \texttt{if}\ e\ \{S'\}\ \texttt{else}\ \{S''\}; S), Q) \to (\Gamma, (\sigma, S'; S), Q)}\ \texttt{<IF-THEN>}$$

$$\frac{\Gamma(e\sigma) = \mathit{false}}{(\Gamma, (\sigma, \texttt{if}\ e\ \{S'\}\ \texttt{else}\ \{S''\}; S), Q) \to (\Gamma, (\sigma, S''; S), Q)}\ \texttt{<IF-ELSE>}$$

$$\frac{\Gamma(e\sigma) = \mathit{true}}{(\Gamma, (\sigma, \texttt{while}\ e\ \{S'\}; S), Q) \to (\Gamma, (\sigma, S'; \texttt{while}\ e\ \texttt{do}\ \{S'\}; S), Q)}\ \texttt{<WHILE-TRUE>}$$

$$\frac{\Gamma(e\sigma) = \mathit{false}}{(\Gamma, (\sigma, \texttt{while}\ e\ \{S'\}; S), Q) \to (\Gamma, (\sigma, S), Q)}\ \texttt{<WHILE-FALSE>}$$

$$\frac{\Gamma(e\sigma) = \mathit{true}}{(\Gamma, (\sigma, \texttt{await}\ e; S), Q) \to (\Gamma, (\sigma, S), Q)}\ \texttt{<AWAITB-TRUE>}$$

$$\frac{\Gamma(e\sigma) = \mathit{false} \qquad (\sigma', S') \in Q}{(\Gamma, (\sigma, \texttt{await}\ e; S), Q) \to (\Gamma, (\sigma', S'), (Q \cup \{(\sigma, \texttt{await}\ e; S)\}) \backslash \{(\sigma', S')\})}\ \texttt{<AWAITB-FALSE>}$$

$$\frac{}{(\Gamma, (\sigma, \texttt{await}\ x?; S), Q) \to (\Gamma, (\sigma, S), Q)}\ \texttt{<AWAITF-SKIP>}$$

$$\frac{(\sigma', S') \in Q}{(\Gamma, (\sigma, \texttt{await}\ x?; S), Q) \to (\Gamma, (\sigma', S'), (Q \cup \{(\sigma, \texttt{await}\ \mathit{true}; S)\}) \backslash \{(\sigma', S')\})}\ \texttt{<AWAITF-SCHED>}$$

$$\frac{(\sigma', S') \in Q}{(\Gamma, (\sigma, \texttt{return}\ e), Q) \to (\Gamma, (\sigma', S'), Q \backslash \{(\sigma', S')\})}\ \texttt{<RETURN>}$$

Figure 6.1: Concrete transition relation

describes a side effect-free assignment. Here, and in the sequel, $e\sigma$ denotes the result of replacing any local variable x in e by $\sigma(x)$. By $\Gamma(e)$ we denote the extension of the variable assignment Γ to the evaluation of the expression e. By $\Gamma[x = v]$, for some value v, we denote the result of updating the value of x in Γ by v.

The rule <NEW> describes the non-deterministic selection of a fresh reference not appearing in the set $\Gamma(\mathit{refs})$. The rule <GET-VALUE> models an assignment involving a get operation on a future variable y which holds a value of some built-in data type by an assignment of a random value $v \in \mathit{Val}$ (of the appropriate type). The rule <GET-REF> models an assignment involving a get operation on a future variable y which holds a reference by first adding a random value $r \in R$ to the set $\Gamma(\mathit{refs})$ and then assign it to the variable x (note that we do *not* exclude that $r \in \Gamma(\mathit{refs})$).

It should be observed that we model the local behavior of an actor. The absence of information about the return values in the semantics of a get operation is accounted for by a non-deterministic selection of an arbitrary return value. Further, since we restrict to the analysis of local deadlocks, we also abstract from the possibility that the get operation blocks and assume that the return value is generated.

The rules regarding choice and iteration statements are standard. The rule <REMOTE-CALL> describes an assignment involving an external call ($\Gamma(y\sigma) \neq \Gamma(\mathbf{this})$, where $y\sigma$ denotes y, if y is a global variable, otherwise it denotes the variable $\sigma(y)$). It is modeled by the creation and storage of a new future reference uniquely identifying the method invocation. On the other hand, according to the rule <LOCAL-CALL> a local call ($\Gamma(y\sigma) = \Gamma(\mathbf{this})$) generates a new process and future corresponding to the method invocation. Also it is checked that the renaming σ' is fresh. Further, by $\Gamma[\bar{z}\sigma' = \Gamma(\bar{e}\sigma)]$ we denote the simultaneous update of Γ which assigns to each local variable $\sigma'(z_i)$ (i.e., the renamed formal parameter z_i) the value of the corresponding actual parameter e_i with its local variables renamed by σ, i.e., the local context of the calling method invocation. For technical convenience we omitted the initialization of the local variables that are not formal parameters. The body of method m is denoted by S'.

The rule <AWAITB-TRUE> describes that when the Boolean condition of the await statement is true, the active process proceeds with the continuation, and <AWAITB-FALSE> describes that when the Boolean condition of the await statement is false, a process is selected for execution. This can give rise to the activation of a disabled process, which is clearly not optimal. The transition system can be extended to only allow the activation of enabled processes. However, this does not affect the results of this chapter and therefore is omitted for notational convenience.

The rule <AWAITF-SKIP> formalizes the assumption that the return value referred to by x has been generated. On the other hand, <AWAITF-SCHED> formalizes the assumption that the return value has not (yet) been generated. Note that we transform the initial await statement into an await on the Boolean condition "true". Availability of the return value then is modeled by selecting the process for

execution. Finally, in the rule RETURN we assume that the return statement is the last statement to be executed. Note that here we do not store the generated return value (see also the discussion in section 6.8).

In view of the above, we have the following definition of a local deadlock.

Definition 6.3.1. *A local configuration* (Γ, p, Q) deadlocks *if*

> *for all* $(\sigma, S) \in \{p\} \cup Q$ *we have that the initial statement of* S *is an await statement* `await` e *such that* $\Gamma(e\sigma) = \mathit{false}$.

In the sequel we describe a method for establishing that an initial configuration does not give rise to a local deadlock configuration. Here it is worthwhile to observe that the above description of the local behavior of a single actor provides an over-approximation of its actual local behavior as part of *any* system of actors. Consequently, absence of a local deadlock of this over-approximation implies absence of a local deadlock in *any* system of actors.

6.4 The Abstract System

Our method of deadlock detection is based on predicate abstraction. This boils down to using predicates instead of concrete value assignments. For the class C, the set $\mathit{Pred}(m)$ includes all (the negations of) the Boolean conditions appearing in the body of m. Further, $\mathit{Pred}(m)$ includes all (negations of) equations $x = y$ between reference variables x and y, where both x and y are global variables of the class C (including `this`) or local variables of m (a reference variable is either a future variable or used to refer to an actor.) In addition to these conditions, the set $\mathit{Pred}(m)$ can also include user-defined predicates that possibly increases the precision of the analysis.

An abstract configuration α is of the form (T, p, Q), where, as in the previous section, p is the active process and Q is a set of suspended processes. The set T provides for each invocation of a method m a logical description of the relation between its local variables and the global variables. Formally, T is a set of pairs (σ, u), where $u \subseteq \mathit{Pred}(m)$, for some method m, is a set of predicates of m with fresh local variables as specified by σ. We assume that for each process $(\sigma, S) \in \{p\} \cup Q$ there exists a corresponding pair $(\sigma, u) \in T$. If for some $(\sigma, u) \in T$ there does not exist a corresponding process $(\sigma, S) \in \{p\} \cup Q$ then the process has terminated. Further, we assume that for any σ there is at most one $(\sigma, u) \in T$ and at most one $(\sigma, S) \in \{p\} \cup Q$.

We next define a transition relation on abstract configurations in terms of a strongest postcondition calculus. To describe this calculus, we first introduce the following notation. Let $L(T)$ denote the set $\{\, u\sigma \,|\, (\sigma, u) \in T \,\}$, where $u\sigma = \{\, \varphi\sigma \,|\, \varphi \in u \,\}$, and $\varphi\sigma$ denotes the result of replacing every local variable x in φ with $\sigma(x)$.

Logically, we view each element of $L(T)$ as a conjunction of its predicates. Therefore, when we write $L(T) \vdash \varphi$, i.e., φ is a logical consequence (in first-order logic) of $L(T)$, the sets of predicates in $L(T)$ are interpreted as conjunctions. (It is worthwhile to note that in practice the notion of logical consequence will also involve the first-order theories of the underlying data structures.) The strongest postcondition, defined below, describes for each basic assignment a and local context $\sigma \in \Sigma(m)$, the set $sp_\sigma(L(T), a)$ of predicates $\varphi \in \mathit{Pred}(m)$ such that $\varphi\sigma$ holds after the assignment, assuming that all predicates in $L(T)$ hold initially.

For an assignment $x = e$ we define the strongest postcondition by

$$sp_\sigma(L(T), x = e) = \{\, \varphi \mid L(T) \vdash \varphi\sigma[e/x],\, \varphi \in \mathit{Pred}(m)\}$$

where $[e/x]$ denotes the substitution which replaces occurrences of the variable x by the side effect-free expression e. For an assignment $x = \texttt{new}$ we define the strongest postcondition by

$$sp_\sigma(L(T), x = \texttt{new}) = \{\, \varphi \mid L(T) \vdash \varphi\sigma[\texttt{new}/x],\, \varphi \in \mathit{Pred}(m)\,\}$$

The substitution $[\texttt{new}/x]$ replaces every equation $x = y$, with y distinct from x, by *false*, $x = x$ by *true*. It is worthwhile to note that for every future variable and variable denoting an actor, these are the only possible logical contexts consistent with the programming language. (Since the language does not support de-referencing, actors encapsulate their local state.)

For an assignment $x = y.\texttt{get}$ we define the strongest postcondition by

$$sp_\sigma(L(T), x = y.\texttt{get}) = \{\, \varphi \mid L(T) \vdash \forall x.\varphi\sigma,\, \varphi \in \mathit{Pred}(m)\,\}$$

The universal quantification of the variable x models a non-deterministic choice for the value of x.

Table 6.2 presents the structural operational semantics of the transition relation for abstract configurations. In the <ASSIGN> rule the set of predicates u for each $(\sigma', u) \in T$, is updated by the strongest postcondition $sp_{\sigma'}(L(T), (x = e)\sigma)$. Note that by the substitution theorem of predicate logic, we have for each predicate φ of this strongest postcondition that $\varphi\sigma'$ will hold after the assignment $(x = e)\sigma$ (i.e., $x\sigma = e\sigma$) because $L(T) \vdash \varphi\sigma[e/x]$. Similarly, the rules <GET> and <NEW> update T of the initial configuration by their corresponding strongest postcondition as defined above.

In the rule <REMOTE-CALL> we identify a remote call by checking whether the information $\texttt{this} \neq y\sigma$ can be added consistently to $L(T)$. By $T \cup \{(\sigma, \varphi)\}$ we denote the set $\{\, (\sigma', u) \in T \mid \sigma' \neq \sigma\,\} \cup \{\, (\sigma, u \cup \{\varphi\}) \mid (\sigma, u) \in T\,\}$. In the rule <LOCAL-CALL> the set of predicates u of the generated invocation of method m consists of all those predicates $\varphi \in \mathit{Pred}(m)$ such that $L(T) \vdash \varphi[\bar{e}\sigma/\bar{z}]$, where $\bar{z}$ denotes the formal parameters of m. By the substitution theorem of predicate

$$\texttt{<ASSIGN>}\quad \frac{T' = \{\, (\sigma', sp_{\sigma'}(L(T), (x = e)\sigma)) \mid (\sigma', u) \in T \,\}}{(T, (\sigma, x = e; S), Q) \to (T', (\sigma, S), Q)}$$

$$\texttt{<GET>}\quad \frac{T' = \{\, (\sigma', sp_{\sigma'}(L(T), (x = y.\texttt{get})\sigma)) \mid (\sigma', u) \in T \,\}}{(T, (\sigma, x = y.\texttt{get}; S), Q) \to (T', (\sigma, S), Q)}$$

$$\texttt{<NEW>}\quad \frac{T' = \{\, (\sigma', sp_{\sigma'}(L(T), (x = \texttt{new})\sigma)) \mid (\sigma', u) \in T \,\}}{(T, (\sigma, x = \texttt{new}; S), Q) \to (T', (\sigma, S), Q)}$$

$$\texttt{<REMOTE-CALL>}\quad \frac{L(T) \cup \{\texttt{this} \neq y\sigma\} \not\vdash \mathit{false}}{(T, (\sigma, x = y!m(\bar{e}); S), Q) \to (T \cup \{(\sigma, \texttt{this} \neq y)\}\}, (\sigma, x = \texttt{new}; S), Q)}$$

$$\texttt{<LOCAL-CALL>}\quad \frac{L(T) \cup \{\texttt{this} = y\sigma\} \not\vdash \mathit{false} \qquad u = \{\, \varphi \mid L(T) \vdash \varphi[\bar{e}\sigma/\bar{z}],\ \varphi \in \mathit{Pred}(m) \,\} \quad \mathit{fresh}(\sigma')}{\begin{array}{c}(T, (\sigma, x = y!m(\bar{e}); S), Q) \to \\ (T \cup \{(\sigma', u)\} \cup \{(\sigma, \texttt{this} = y)\}, (\sigma, x = \texttt{new}; S), Q \cup \{(\sigma', S')\})\end{array}}$$

$$\texttt{<IF-THEN>}\quad \frac{L(T) \cup \{e\sigma\} \not\vdash \mathit{false}}{\begin{array}{c}(T, (\sigma, \texttt{if}\ e\ \{S'\}\ \texttt{else}\ \{S''\}; S), Q) \\ \to (T \cup \{(\sigma, e)\}, (\sigma, S'; S), Q)\end{array}}$$

$$\texttt{<IF-ELSE>}\quad \frac{L(T) \cup \{\neg e\sigma\} \not\vdash \mathit{false}}{\begin{array}{c}(T, (\sigma, \texttt{if}\ e\ \{S'\}\ \texttt{else}\ \{S''\}; S), Q) \\ \to (T \cup \{(\sigma, \neg e)\}, (\sigma, S''; S), Q)\end{array}}$$

$$\texttt{<WHILE-TRUE>}\quad \frac{L(T) \cup \{e\sigma\} \not\vdash \mathit{false}}{\begin{array}{c}(T, (\sigma, \texttt{while}\ e\ \texttt{do}\ \{S\}'; S), Q) \\ \to (T \cup \{(\sigma, e)\}, (\sigma, S'; \texttt{while}\ e\ \texttt{do}\ \{S'\}; S), Q)\end{array}}$$

$$\texttt{<WHILE-FALSE>}\quad \frac{L(T) \cup \{\neg e\sigma\} \not\vdash \mathit{false}}{\begin{array}{c}(T, (\sigma, \texttt{while}\ e\ \texttt{do}\ \{S'\}; S), Q) \\ \to (T \cup \{(\sigma, \neg e)\}, (\sigma, S), Q)\end{array}}$$

$$\texttt{<AWAIT-TRUE>}\quad \frac{L(T) \cup \{e\sigma\} \not\vdash \mathit{false}}{(T, (\sigma, \texttt{await}\ e; S), Q) \to (T \cup \{(\sigma, e)\}, (\sigma, S), Q)}$$

$$\texttt{<AWAIT-FALSE>}\quad \frac{L(T) \cup \{\neg e\sigma\} \not\vdash \mathit{false} \qquad (\sigma', S') \in Q}{(T, (\sigma, \texttt{await}\ e; S), Q) \to (T \cup \{(\sigma, \neg e)\}, (\sigma', S'), (Q \cup \{(\sigma, \texttt{await}\ e; S)\})\backslash\{(\sigma', S')\})}$$

$$\texttt{<AWAITF-SKIP>}\quad (T, (\sigma, \texttt{await}\ x?; S), Q) \to (T, (\sigma, S), Q)$$

$$\texttt{<AWAITF-SCHED>}\quad \frac{(\sigma', S') \in Q}{\begin{array}{c}(T, (\sigma, \texttt{await}\ x?; S), Q) \to \\ (T, (\sigma', S'), (Q \cup \{(\sigma, \texttt{await}\ \mathit{true}; S)\})\backslash\{(\sigma', S')\})\end{array}}$$

$$\texttt{<RETURN>}\quad \frac{(\sigma', S') \in Q}{(T, (\sigma, \texttt{return}\ e), Q) \to (T, (\sigma', S'), Q\backslash\{(\sigma', S')\})}$$

Figure 6.2: Abstract transition system

logic, the (simultaneous) substitution $[\bar{e}\sigma/\bar{z}]$ ensures that φ holds for the generated invocation of method m. Note that by definition, $L(T)$ only refers to fresh local variables, i.e., the local variables of m do not appear in $L(T)$ because for any $(\sigma, u) \in T$ we have that $\sigma(x)$ is a fresh variable not appearing in the given class C. For technical convenience we omitted the substitution of the local variables that are not formal parameters. The renaming σ', which is assumed not to appear in T, introduces fresh local variable names for the generated method invocation. The continuation S' of the new process is the body of method m. The generation of a new future in both the rules <REMOTE-CALL> and <LOCAL-CALL> is simply modeled by the $x = \texttt{new}$ statement.

By <IF-THEN>, the active process transforms to the "then" block, i.e. S', followed by S, if the predicate set $L(T)$ is consistent with the guard e of the if-statement. (Note that as $L(T)$ is in general not complete, it can be consistent with e as well as with $\neg e$.) The other rules regarding choice and iteration statements are defined similarly. By <RETURN> the active process terminates, and is removed from the configuration. A process is selected from Q for execution. Note that the pair $(\sigma, u) \in T$ is not affected by this removal.

The rules <AWAIT-TRUE> and <AWAIT-FALSE> specify transitions assuming the predicate set $L(T)$ is consistent with the guard e and with $\neg e$, respectively. In the former case, the `await` statement is skipped and the active process continues, whereas in the latter, the active process releases control and a process from Q is activated. Similar to the concrete semantics in the previous section, in <AWAITF-SKIP> and <AWAITF-SCHED>, the active process non-deterministically continues or cooperatively releases the control. In the latter, a process from Q is activated.

We conclude this section with the counterpart of Definition 6.3.1 for the abstract setting.

Definition 6.4.1. *A local configuration (T, p, Q) is a (local)* deadlock *if*

> *for all $(\sigma, S) \in \{p\} \cup Q$ we have that the initial statement of S is an await statement* `await` e *such that $L(T) \cup \{\neg e\sigma\} \nvdash$ false.*

6.5 Correctness of Predicate Abstraction

In this section we prove that the concrete system is simulated by the abstract system. To this end we introduce a simulation relation $\sim$ between concrete and abstract configurations:

$$(\Gamma, p, Q) \sim (T, p, Q), \text{ if } \Gamma \models L(T)$$

where $\Gamma \models L(T)$ denotes that Γ satisfies the formulas of $L(T)$.

Theorem 3. *The abstract system is a simulation of the concrete system.*

Proof. Given $(\Gamma, p, Q) \sim (T, p, Q)$ and a transition $(\Gamma, p, Q) \to (\Gamma', p', Q')$, we need to prove that there exists a transition $(T, p, Q) \to (T', p', Q')$ such that $(\Gamma', p', Q') \sim (T', p', Q')$.

For all the rules that involve the evaluation of a guard e, it suffices to observe that $\Gamma \models L(T)$ and $\Gamma \models e$ implies $L(T) \cup \{e\} \not\vdash \mathit{false}$.

We treat the case $x = e$ where e is a side effect-free expression (the others cases are treated similarly). If $p = (\sigma, x = e; S)$, where e is a side effect-free expression, then $\Gamma' = \Gamma[(x = e)\sigma]$. We put $T' = \{\, (\sigma', sp_{\sigma'}(L(T), (x = e)\sigma)) \mid (\sigma', u) \in T \,\}$. Then it follows that $(T, p, Q) \to (T', p', Q')$. To prove $\Gamma' \models L(T')$ it remains to show for $(\sigma, u) \in T$ and $\varphi \in sp_{\sigma'}(L(T), (x = e)\sigma)$ that $\Gamma' \models \varphi\sigma'$: Let $(\sigma', u) \in T$ and $\varphi \in sp_{\sigma'}(L(T), (x = e)\sigma)$. By definition of the strongest postcondition, we have $L(T) \vdash \varphi\sigma'[(x = e)\sigma]$. Since $\Gamma \models L(T)$, we have $\Gamma \models \varphi\sigma'[(x = e)\sigma]$. Since $\Gamma' = \Gamma[x\sigma = \Gamma(e\sigma)]$, we obtain from the substitution theorem of predicate logic that

$$\Gamma' \models \varphi\sigma' \iff \Gamma \models \varphi\sigma'[(x = e)\sigma]$$

and hence we are done. □

We conclude this section with the following observation: if the initial abstract configuration (T, p, Q) does not give rise to a local deadlock then also the configuration (Γ, p, Q) does not give rise to a local deadlock, when $\Gamma \models L(T)$. To see this, by the above theorem it suffices to note that if (Γ', p', Q') is a local deadlock and $\Gamma' \models L(T')$ then (T', p', Q') is also a local deadlock because for any $(\sigma, \texttt{await}\ e; S) \in \{p'\} \cup Q'$ we have that $\Gamma' \not\models e\sigma$ implies $L(T') \cup \{\neg e\sigma\} \not\vdash \mathit{false}$.

6.6 Example

We represent the proposed method by means of an example. Given partial definition of the class C as follows:

```
class C {
  Int a = 0;
  void m() {
    a= a + 1; await a < 5;
  }
  ...
}
```

We want to check the program for the absence of the local deadlock, where Q contains only one process $(\sigma, \texttt{a=a+1;await a<5})$, which is an invocation of the method m. The $\mathit{Pred}(m) = \{a < 5, \neg(a < 5), a = 3, \neg(a = 3)\}$ is the set of predicates of the method m, and a user-defined predicate $a = 3$ and its negation. We try to check the system for the absence of deadlock for the initial $u = \{a < 5\}$,

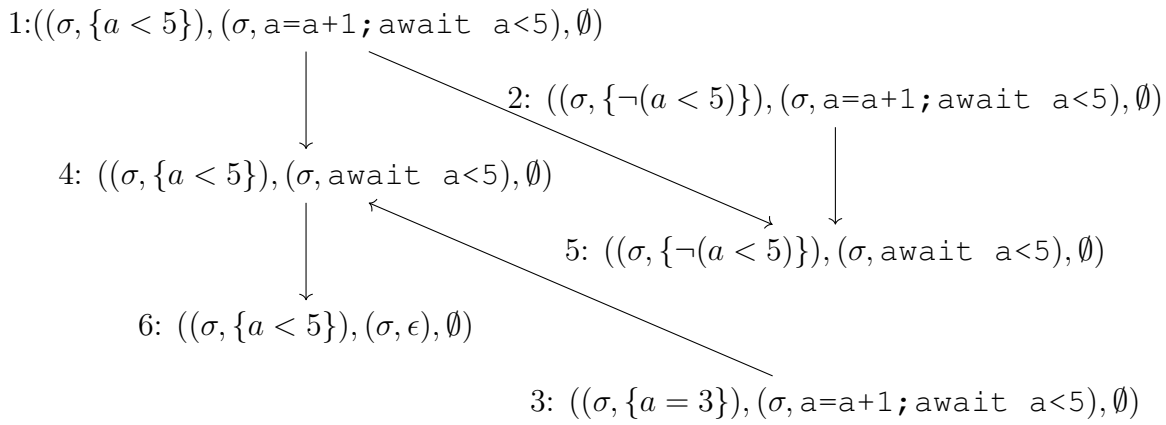

Figure 6.3: Example of the abstract system

$u = \{\neg(a < 5)\}$ and $u = \{a = 3\}$. The abstract systems for three different initial states form a finite transition system which is shown in Figure 6.3.

The states 1,2 and 3 denote the three initial states. State 5 denotes a local deadlock configuration. State 6 denotes a normal termination configuration. The resulting transition system shows that the execution of method m, where initially $a \geqslant 4$, possibly causes deadlock.

6.7 Decidability of Deadlock Detection

The abstract local behavior of a single actor, as defined in the previous section, gives rise, for a given initial configuration, to an infinite transition system because of dynamic generation of local calls and the corresponding introduction of fresh local variables. In this section we show how we can model an abstract system for which the transition relation is computable as well-structured transition system and obtain the decidability of deadlock detection for such abstract systems. To this end, we first provide a canonical representation of an abstract configuration which abstracts from renamings of the local variables by means of *multisets* of *closures*. A closure of a method m is a pair (u, S), where S is a *continuation* of the body of m and $u \subseteq \mathit{Pred}(m)$. (Here $\mathit{Pred}(m)$ denotes the set of predicates associated with m as defined in Section 6.3). The set of continuations of a statement S is the smallest set $\mathit{Cont}(S)$ such that $S \in \mathit{Cont}(S)$ and $\epsilon \in \mathit{Cont}(S)$, where the "empty" statement ϵ denotes termination, and which is closed under the following conditions

- $S'; S'' \in \mathit{Cont}(S)$ implies $S'' \in \mathit{Cont}(S)$
- `if` e $\{S_1\}$ `else` $\{S_2\}$; $S' \in \mathit{Cont}(S)$ implies $S_1; S' \in \mathit{Cont}(S)$ and $S_2; S' \in \mathit{Cont}(S)$
- `while` e $\{S'\}$; $S'' \in \mathit{Cont}(S)$ implies S'; `while` e $\{S'\}$; $S'' \in \mathit{Cont}(S)$.

Note that for a given method the set of all possible closures is finite. We formally represent a multiset of closures as a function which assigns a natural number $f(c)$ to each closure c which indicates the number of occurrences of c. For notational convenience we write $c \in f$ in case $f(c) > 0$.

In preparation of the notion of canonical representation of abstract configurations, we introduce for every abstract configuration $\alpha = (T, p, Q)$ the set $\bar{\alpha}$ of triples (σ, u, S) for which $(\sigma, u) \in T$ and either $(\sigma, S) \in \{p\} \cup Q$ or $S = \epsilon$.

Definition 6.7.1. *An abstract configuration (T, p, Q) is canonically represented by a multiset of closures f, if for every method m and closure (u, S) of m we have*

$$f((u, S)) = |\{\, \sigma \mid (\sigma, u, S) \in \bar{\alpha} \,\}|$$

(where $|V|$ denotes the cardinality of the set V).

Note that each abstract configuration has a unique multiset representation. For any multiset f of closures, let $T(f)$ denote the set of predicates $\{\exists v \mid (v, S)^n \in f\}$, where $\exists v$ denotes v with all the local variables appearing in the conjunction of the predicates of v existentially quantified.

The following lemma states the equivalence of a set of closures and its canonical representation.

Lemma 6.7.1. *Let the abstract configuration (T, p, Q) be canonically represented by the multiset of closures f. Further, let $(\sigma, u) \in T$, where $\sigma \in \Sigma(m)$, and $\varphi \in Pred(m)$. It holds that*

$$L(T) \vdash \varphi\sigma \text{ iff } \{u\} \cup T(f) \vdash \varphi$$

Proof. Proof-theoretically we reason, in first-order logic, as follows. For notational convenience we view a set of predicates as the conjunction over its elements. By the Deduction Theorem we have

$$L(T) \vdash \varphi\sigma \text{ iff } \vdash L(T) \to \varphi\sigma$$

From the laws of universal quantification we obtain

$$\vdash L(T) \to \varphi\sigma \text{ iff } \vdash \forall X(L(T) \to \varphi\sigma)$$

and

$$\vdash \forall X(L(T) \to \varphi\sigma) \text{ iff } \vdash \exists X L(T) \to \varphi\sigma$$

where X denotes the set of local variables appearing in $L(T) \setminus \{u\sigma\}$. Note that no local variable of X appears in $\varphi\sigma$ or $u\sigma$.

Since any two distinct $v, v' \in L(T)$ have no local variables in common, we can

push the quantification of $\exists X L(T)$ inside. That is,

$$\vdash \exists X L(T) \to \varphi\sigma \text{ iff } \vdash \{\, \exists X v \mid v \in L(T) \,\} \to \varphi\sigma$$

No local variable of X appears in $u\sigma$, therefore we have

$$\vdash \{\, \exists X v \mid v \in L(T) \,\} \to \varphi\sigma \text{ iff } \vdash u\sigma \wedge \{\, \exists X v \mid v \in L(T) \,\} \to \varphi\sigma$$

Again by the Deduction Theorem we then have

$$\vdash u\sigma \wedge \{\, \exists X v \mid v \in L(T) \,\} \to \varphi\sigma \text{ iff } \{u\sigma\} \vdash \{\, \exists X v \mid v \in L(T) \,\} \to \varphi\sigma$$

Clearly $u\sigma \vdash \exists u$ and $\exists X v$ is logically equivalent to $\exists v$, for any $v \in L(T) \setminus \{u\sigma\}$. So, we have

$$\{u\sigma\} \vdash \{\, \exists X v \mid v \in L(T) \,\} \to \varphi\sigma \text{ iff } \{u\sigma\} \vdash \{\, \exists v \mid v \in L(T) \,\} \to \varphi\sigma$$

Since f represents (T, p, Q) we have that $T(f) = \{\, \exists v \mid v \in L(T) \,\}$. Renaming the local variables of $u\sigma$ and $\varphi\sigma$ then finally gives us

$$\{u\sigma\} \vdash \{\, \exists v \mid v \in L(T) \,\} \to \varphi\sigma \text{ iff } \{u\} \vdash T(f) \to \varphi$$

which proves the lemma. □

We next define an ordering on multisets of closures.

Definition 6.7.2. *By $f \preccurlyeq f'$ we denote that $f(c) \leqslant f'(c)$ and $f'(c) = 0$ if $f(c) = 0$.*

In other words, $f \preccurlyeq f'$ if all occurrences of f belong to f' and f' does not add occurrences of closures which do not already occur in f. The following result states that this relation is a well-quasi-ordering.

Lemma 6.7.2. *The relation $f \preccurlyeq f'$ is a quasi-ordering such that for any infinite sequence $(f_n)_n$ there exist indices $i < j$ such that $f_i \preccurlyeq f_j$.*

Proof. First observe that for a given class there is only a finite number of closures. We show that the proof for the standard subset relation for multisets also holds for this variation. Assume that for some set X of closures we have constructed an infinite subsequence $(f'_n)_n$ of $(f_n)_n$ such that $f'_i(c) \leqslant f'_j(c)$, for every $c \in X$ and $i < j$. Suppose that for every $c \notin X$ the set $\{\, k \mid f'_j(c) = k,\, j \in \mathbb{N} \,\}$ is bounded. It follows that there exists an f'_k which appears infinitely often in $(f'_n)_n$, since there exists only a finite number of combinations of occurrences of closures in $\bar{X} = \{\, c \mid c \notin X \,\}$. On the other hand, if there exists a $d \notin X$ such that set $\{\, k \mid f'_j(d) = k,\, j \in \mathbb{N} \,\}$ has no upperbound then we can obtain a subsequence $(f''_n)_n$ of $(f'_n)_n$ such that $f''_i(c) \leqslant f''_j(c)$ for every $c \in X \cup \{d\}$ and $i < j$. Thus, both cases lead to the existence of indices $i < j$ such that $f_i \preccurlyeq f_j$. □

From the above lemma it follows immediately that the following induced ordering on abstract configurations is also a well-quasi-ordering.

Definition 6.7.3. *We put* $(T, (\sigma, S), Q) \preccurlyeq (T', (\sigma', S), Q')$ *iff* $f \preccurlyeq f'$*, for multi-sets of closures* f *and* f' *(uniquely) representing* $(T, (\sigma, S), Q)$ *and* $(T', (\sigma', S), Q')$*, respectively.*

We can now formulate and prove the following theorem which states that this well-quasi-ordering is preserved by the transition relation of the abstract system.

Theorem 4. *For abstract configurations* α*,* α'*, and* β*, if* $\alpha \to \alpha'$ *and* $\alpha \preccurlyeq \beta$ *then* $\beta \to \beta'$*, for some abstract configuration* β' *such that* $\alpha' \preccurlyeq \beta'$*.*

Proof. The proof proceeds by a case analysis of the transition $\alpha \to \alpha'$. Crucial in this analysis is the observation that $\alpha \preccurlyeq \beta$ implies that $\alpha = (T, p, Q)$ and $\beta = (T', p, Q)$, for some T and T' such that

$$L(T) \vdash \varphi\sigma \iff L(T') \vdash \varphi\sigma'$$

for renamings $\sigma, \sigma' \in \Sigma(m)$, where m is a method defined by the given class C, such that $(\sigma, u, S) \in \bar{\alpha}$ and $(\sigma', u, S) \in \bar{\beta}$, for some closure (u, S) and predicate φ of the method m. This follows from Lemma 6.7.1 and that $f \preccurlyeq f'$ implies $T(f) = T(f')$, where f and f' represent α and β, respectively. Note that by definition, f' does not add occurrences of closures which do not already occur in f. □

It follows that abstract systems for which the transition relation is computable are well-structured transition systems (see [36] for an excellent explanation and overview of well-structured transition systems). For such systems the *covering* problem is decidable. That is, for any two abstract configurations α and β it is decidable whether starting from α it is possible to cover β, meaning, whether there exists a computation $\alpha \to^* \alpha'$ such that $\beta \preccurlyeq \alpha'$. To show that this implies decidability of absence of deadlock, let α be a *basic* (abstract) deadlock configuration if α is a deadlock configuration according to Definition 6.4.1 and for any closure (u, S) there exists *at most one renaming* σ such that $(\sigma, u, S) \in \bar{\alpha}$. Note that thus $f(c) = 1$, for any closure c, where f represents α. Let Δ denote the set of all basic deadlock configurations. Note that this is a finite set. Further, for every (abstract) deadlock configuration α there exists a basic deadlock configuration $\alpha' \in \Delta$ such that $f \preccurlyeq f'$, where f and f' represent α and α', respectively. This is because the different renamings of the same closure do not affect the definition of a deadlock. Given an initial abstract configuration α, we now can phrase presence of deadlock as the covering problem of deciding whether there exists a computation starting from α reaching a configuration β that covers a deadlock configuration in Δ.

Summarizing the above, we have the following the main technical result of this chapter.

Theorem 5. *Given an abstract system with a computable transition relation and an abstract configuration α, it is decidable whether*

$$\{\, \beta \mid \alpha \rightarrow^* \beta \,\} \cap \{\, \beta \mid \exists \beta' \in \Delta \colon \beta' \preccurlyeq \beta \,\} = \emptyset \tag{6.1}$$

Given this result and the correctness of predicate abstraction, to show that an initial concrete configuration (Γ, p, Q) does *not* give rise to a local deadlock, it suffices to construct an abstract configuration $\alpha = (T, p, Q)$ such that $\Gamma \models L(T)$ and for which Equation (6.1) holds. Note that we can construct T by the constructing pairs (σ, u), where $u = \{\phi \in \mathit{Pred}(m) \mid \Gamma \models \phi\sigma\}$ (assuming that $\sigma \in \Sigma(m)$).

6.8 Conclusion

For future work we first have to validate our method for detecting local deadlock in tool-supported case studies. For this we envisage the use of the theorem-prover KeY [3] for the construction of the abstract transition relation, and its integration with *on-the-fly* reachability analysis of the abstract transition system.

Another major challenge is the extension of our method to (predicate) abstraction of *local* futures, that is, futures generated by self calls. Note that in the method described in the present chapter, we do not distinguish between these futures and those generated by external calls. The main problem is to extend the abstraction method to describe and reason about local futures which preserves the properties of a well-structured transition system.

Of further interest, in line with the above, is the integration of the method of predicate abstraction in the theorem-prover KeY for reasoning *compositionally* about general safety properties of actor-based programs. For reasoning about programs in the ABS language this requires an extension of our method to *synchronous* method calls and *concurrent object groups*.