Detecting Deadlocks in Formal System Models with Condition Synchronization

Eduard Kamburjan

Department of Computer Science, Technische Universität Darmstadt, Germany
kamburjan@cs.tu-darmstadt.de

Abstract. We present a novel notion of deadlock for synchronization on arbitrary boolean conditions and a sound, fully automatic deadlock analysis. Contrary to other approaches, our analysis aims to detect deadlocks caused by faulty system design, rather than implementation bugs. We analyze synchronization on boolean conditions on the fields of an object instead of targeting specific synchronization primitives. As usual, a deadlock is a circular dependency between multiple tasks. A task depends on a second task if the execution of this second task has a side-effect that makes the blocking guard-condition of the first one evaluate to true. This requires an analysis of the computations in a method beyond syntactic properties and we integrate a logical validity calculus to do so.

1 Introduction

Deadlock is an essential notion of error in distributed systems and is commonly defined as a blocked configuration with *circular dependencies* among multiple tasks. Deadlocks have been examined for different notions of dependency: Resource dependencies are defined between acquire and release actions, or message dependencies, which are defined between receiving and sending actions on channels, and other notions based on other synchronization primitives.

The control flow of a system design or model is described using common synchronization primitives like locks, in most languages. The primitives reduce to one of the aforementioned dependency notions and allow to use dependency analyses for deadlock detection. Systems however are rarely directly designed with concrete primitives in mind – the design makes use of the more abstract synchronization *patterns* with synchronization on conditions. Condition synchronization can be expressed with, e.g., a statement **await** `i>0`, which suspends the active task until the guard-condition `i>0` becomes true. Such a statement is not available in most languages, but can be seen as an abstraction to the established conditional variables pattern with monitors and loops. The direct use of a condition synchronization statement is nearer to the modelers intention of when a task will resume. Condition synchronization can be compiled into low-level synchronization primitives, but they do not trivially reduce to resources or messages: the most common compilation into condition variables requires the addition of new monitors, locks and loops.

We propose a formalization of dependency that fits the intuition of the system designer better than purely syntactical approaches or approaches based on translation into low-level synchronization primitives. A task t_1 depends on t_2 if the continuation of t_2 would make the guard-condition $\mathtt{b}$ of t_1 true at a point where t_1 may be scheduled. This notion of dependency requires to evaluate the guard $\mathtt{b}$ and to analyze all side-effects of the continuation of t_2.

The analysis builds a dependency graph: First, for each field new dependencies are added from each read in a condition synchronization to each write. In the second step, all those edges are removed, for which we can show that the execution of the writing method will *never* make the guard in question true.

The actor-based, object-oriented Abstract Behavioral Specification (ABS) modeling language [12] implements an **await** statement. But all three deadlock analysis tools developed for ABS [8–10] do not consider dependencies introduced by **await** statements. Motivated by this shortcoming, we implemented our approach for ABS, while the theory can be applied to other languages.

Our implementation extends the DECO tool [8] in the Static Analysis for Active Objects (SACO) toolsuite, and integrates the KeY-ABS theorem prover [5] to discard dependencies.

We evaluate our approach on industrial case studies, which show that the precision depends on the communication structure and the complexity of required SMT-theories for the types occurring in the program. It also shows that our analysis is precise enough for useable feedback in most cases and gives valuable clues to detect deadlocks caused by *errors in the modeled system* rather than errors stemming from the wrong use of synchronization primitives: The number of false positives in all but one case studies is small enough to check all detected potential deadlocks by hand. Our main contributions are (1) a novel notion of dependency and deadlock for condition synchronization and (2) a sound deadlock analysis for full coreABS that integrates a theorem prover into a dependency-based deadlock analysis.

This paper is organized as follows: Section 2 introduces condition synchronization and gives examples of its usage as an abstraction for low-level synchronization primitives. Section 3 introduces a simple language with condition synchronization. Section 4 defines our notion of deadlock and Section 5 describes our analysis for it, Section 6 reports on the implementation and Section 7 concludes with future and related work.

2 On Condition Synchronization in System Design

Our aim is to analyze the control flow of a system *design* to ensure that it does not include circular dependencies. For this, we concentrate on the boolean conditions on which processes may synchronize to achieve the intended control flow. We are not concerned with the usage of low-level primitives like locks: Deadlocks caused by low-level primitives are indications of incorrect usage of the concurrency model, e.g., forgetting to unlock or unlocking twice. Deadlocks

caused by the boolean conditions are signs of *errors in the design*: the program cannot progress because its designed control flow itself contains a bug.

```
1  public void m1() {
2    lock.lock();
3    a = 1;
4    aIsOne.signalAll();
5
6  }
7  public void m2(){
8    lock.lock();
9    while(a != 1) aIsOne.await();
10   a = 0;
11   lock.unlock();
12 }
```

```
1  public void m1() {
2    lock.lock();
3    a = 2; //bug
4    aIsOne.signalAll();
5    lock.unlock();
6  }
7  public void m2(){
8    lock.lock();
9    while(a != 1) aIsOne.await();
10   a = 0;
11   lock.unlock();
12 }
```

Fig. 1. Two Java snippets for condition variables. The methods run in separate threads.

Example 1. Consider the Java code on the left in Fig. 1. If both described methods are running in parallel in two threads on the same object, they need not progress, as `m1` fails to unlock. Now consider the Java code on the right. Again both methods may not progress — however the reason is that `m2` waits for `m1` to change the internal state. The first example is an implementation bug: The lock is used wrongly. The second example is a design bug: the combined control flow of `m1` and `m2` is designed wrongly — `m2` does not continue after `m1`.

Both kinds of errors cannot be sharply distinguished — a wrong usage of synchronization primitives may also be a result of erroneous design, e.g., if unlocking twice is a consequence of the intended control flow. Deadlocks caused by synchronization primitives have been studied extensively [4, 9–11, 20, 22] and focus for the most part on syntactic properties, not information in the guard. In this work we concentrate on condition synchronization. We aim to detect bugs in the design itself, helping the software architect, not the implementing programmer and use this system model: We abstract away from the low-level primitives and only consider the aforementioned **await** statement with a cooperative scheduling concurrency model, where every context switch is explicit.

Condition Synchronization as Abstraction To illustrate the difference between condition synchronization and synchronization via low-level primitives, we show the use of condition synchronization as an abstraction of condition variables. A conditional variable is a predicate associated with a monitor and a lock. All threads waiting for the condition are notified by the monitor once the predicate may become true. If the guard evaluates to false, the notified threads become inactive again.

```
1 public void put(Object x) {
2  lock.lock();
3  while (count >= items.length)
4    notFull.await();
5  //add x to queue here
6  notEmpty.signalAll();
7  lock.unlock();
8 }
```

```
1 put(Object x){
2   await count < items.length;
3  //add x to queue here
4 }
```

Fig. 2. A Java method and its Abstraction

The Java code on the left in Fig. 2 shows the use of condition variables to add an element to a bounded queue, once the queue is not full. Here, the thread waits for the list to be below its maximal capacity. Otherwise it waits on the monitor `notFull` until the state changes and it is notified. If it modifies the state itself, it notifies all threads waiting for the list to be not empty by calling `signalAll` on the monitor `notEmpty`. Deadlock analysis can be performed by analyzing the possible sequences of calls to `await` and `signalAll` [16], as *every* call to `signalAll` causes the process to execute one more loop iteration.

With condition synchronization and cooperative scheduling, we can express this method as shown on the right in Fig. 2. The lock and the monitors are not part of the code, it is thus not necessary to check their correct usage already in the design: The method only switches context at the **await** statement and continues execution once its guard evaluates to true.

Condition Synchronization as a Modeling Tool Condition synchronization is not only a useful tool for modeling, it clarifies reasoning about control flow by abstracting from implementation details. E.g., in the above example the `notEmpty` and `notFull` monitors are not part of the code. This makes it unnecessary to ensure that the correct monitors are used.

Notions of deadlock for condition variables based on the correct use of the involved primitives have two down-sides: First, the condition itself is not determining the dependencies – dependencies are determined by the additional structure the programmer *assumes* to guarantee deadlock freedom. This structure (1) leads to a large overhead, as for each condition a monitor has to be added and (2) adds another layer between the system design and the analyzed artifact. Secondly, in the sketched situation in Java, the waiting thread *may progress*, if another process was active and called `signalAll`; as it must execute the loop to reevaluate its guard-condition. We abstract away from the reevaluation, and assume it is handled by the runtime environment – by abstracting to condition synchronization, the results are nearer to the intuition of the designer.

3 A Language with Boolean Guards and Dynamic Logic

We introduce a simple language SYNC with cooperative scheduling, asynchronous communication and conditional synchronization. SYNC is a simplified version of ABS [12], following the formalization of the semantics in [8]. We ignore futures, which synchronize processes on termination similar to thread joins, and return values for presentation's sake, as those dependencies have been described by Flores-Montoya et al. [8]. Our implementation considers full coreABS.

A SYNC-program is a set of objects and a **main** block. Each object has fields and methods. All objects are running in parallel and share no state. An object may only change its active task, if the active task explicitly releases control. Control can be released by termination or a special statement **await** `b;` which suspends the active task, and allows its reactivation only once the boolean expression `b` evaluates to true. This statement models condition synchronization within one object. Between multiple objects, only asynchronous method calls of the form **async** X.m($\overline{\mathtt{e}}$) are possible. Such an `statement` has been introduced and examined earlier [21], but does not correspond to other `await` concepts. E.g., pthreads implements an `await` function, but usess it to implement barriers, not condition synchronization. As seen in the previous Section, **await** can be compiled into condition variables.

Definition 1 (Syntax). *We underspecify the sets of types and expressions. For the examples, we assume types for booleans, integers, lists and* Object*, as well as the usual operations and literals for their elements. Let* `e` *range over expressions,* T *over types,* `v` *over variable names,* `f` *over field names and* X *over object names.* $\overline{\cdot}$ *denotes possibly empty lists. A program* Prgm *is defined as follows:*

$$\mathsf{Prgm} ::= \overline{\mathsf{O}}\ \textbf{main}\{\mathtt{s}\} \qquad \mathsf{O} ::= \textbf{object}\ \mathsf{X}\ \{\overline{\mathsf{M}}\ \overline{\mathsf{T}\ \mathtt{f}=\mathtt{e}}\} \qquad \mathsf{M} ::= \mathtt{m}(\overline{\mathsf{T}\ \mathtt{v}})\{\mathtt{s}\}$$
$$\mathtt{s} ::= \textbf{async}\ \mathtt{e.m}(\overline{\mathtt{e}}) \mid \mathtt{f}=\mathtt{e} \mid \mathsf{T}\ \mathtt{v}=\mathtt{e} \mid \textbf{await}\ \mathtt{e} \mid \textbf{if}(\mathtt{e})\mathtt{s}\ \textbf{else}\ \mathtt{s}\ \textbf{fi} \mid \textbf{skip} \mid \mathtt{s};\mathtt{s}$$

Example 2. In the following code, the object `Queue` models a queue with maximal length of 5 and the main block pushes a number into the queue and afterwards removes it. It is not guaranteed that the `push` method will *start* to execute first. The synchronization with **await**, however, guarantees that it *terminates* first.

```
1 object Queue{
2   List<Int> list = Nil;
3   push(Int i){ await size(list) < 5; list = [i]::list;}
4   pop(){ await size(list) > 0; list = tail(list);}
5 }
6 main{
7   async Queue!push(1);
8   async Queue!pop();
9 }
```

Definition 2 (Runtime Syntax). *D is the value domain, with* $\{\mathbf{tt}, \mathbf{ff}\} \subseteq D$*. Let* X *range over object names, i over* $\mathbb{N}$*,* `s` *over statements, σ over functions*

$$\frac{}{\begin{array}{c}\mathbf{tsk}(\mathsf{X}, i, \textsf{await } \mathtt{e}, \sigma)\ \mathbf{obj}(\mathsf{X}, i, \rho)\ \mathsf{C} \\ \rightarrow_{(i)}\ \mathbf{tsk}(\mathsf{X}, i, \textsf{await } \mathtt{e}, \sigma)\ \mathbf{obj}(\mathsf{X}, \bot, \rho)\ \mathsf{C}\end{array}}\ (\textsf{wait}) \qquad \frac{[\![\mathtt{e}]\!]_{\sigma,\rho} = \mathbf{tt}}{\begin{array}{c}\mathbf{tsk}(\mathsf{X}, i, \textsf{await } \mathtt{e}, \sigma)\ \mathbf{obj}(\mathsf{X}, \bot, \rho)\ \mathsf{C} \\ \rightarrow_{(i)}\ \mathbf{tsk}(\mathsf{X}, i, \bot, \sigma)\ \mathbf{obj}(\mathsf{X}, i, \rho)\ \mathsf{C}\end{array}}\ (\textsf{cont})$$

$$\frac{[\![\mathtt{e}]\!]_{\sigma,\rho} = \mathsf{X}' \qquad j \text{ does not appear in } \mathsf{C} \qquad \mathsf{C} = \mathbf{obj}(\mathsf{X}', l, \rho')\ \mathbf{obj}(\mathsf{X}, i, \rho)\ \mathsf{C}'}{\begin{array}{c}\mathbf{tsk}(\mathsf{X}, i, \textsf{async } \mathtt{e}.\mathtt{m}(\overline{\mathtt{e}}), \sigma)\ \mathsf{C} \\ \rightarrow_{(i,j)}\ \mathbf{tsk}(\mathsf{X}, i, \bot, \sigma)\ \mathbf{tsk}(\mathsf{X}', j, M(\mathtt{m}), \hat{M}([\![\overline{\mathtt{e}}]\!]_{\sigma,\rho}))\ \mathsf{C}\end{array}}\ (\textsf{call})$$

Fig. 3. Selected Small-Step Operational Semantics Rules

that map variable names to domain elements and ρ over functions that map field names to domain elements. We define configurations C as follows:

$$\mathsf{C} ::= \mathbf{tsk}(\mathsf{X}, i, \mathtt{s}, \sigma) \mid \mathbf{obj}(\mathsf{X}, i, \rho) \mid \mathsf{C}\ \mathsf{C}$$

The composition of configurations is commutative and associative, i.e. $\mathsf{C}\ \mathsf{C}' = \mathsf{C}'\ \mathsf{C}$ and $\mathsf{C}\ (\mathsf{C}'\ \mathsf{C}'') = (\mathsf{C}\ \mathsf{C}')\ \mathsf{C}''$. Well-formedness conditions can be found in [12].

A configuration contains *tasks* and *objects*. An object $\mathbf{obj}(\mathsf{X}, i, \rho)$ has a unique name X, an active task id i and a store ρ. If inactive, the task id is the special symbol $\bot$. A task $\mathbf{tsk}(\mathsf{X}, i, \mathtt{s}, \sigma)$ has a unique id i, a local store σ, the id of its object X and the remaining statement. A terminated task has the special symbol $\bot$ as its statement.

We denote the initial configuration of a program Prgm with $\mathbb{I}(\mathsf{Prgm})$. The definition is straightforward and the main block is running in a special object. We assume that each store ρ is initialized with a special field X_f for each object X with $\rho(\mathsf{X}_f) = \mathsf{X}$. The method body of a method $\mathtt{m}$ is denoted $M(\mathtt{m})$ and the initial local store of a task executing $\mathtt{m}$ with parameters $\overline{d}$ with $\hat{M}(\overline{d})$.

The most important rules are shown in Fig 3: The rule (wait) suspends a process by setting the task id of the corresponding object to $\bot$. The **await** statement is not removed. The rule (cont) removes the **await** statement when reactivating a process – the corresponding object must be inactive and the guard must hold. The rule (call) starts a new process, which is not set as active upon creation.

A configuration is terminated if all tasks and objects have the forms

$$\mathbf{tsk}(\mathsf{X}_i, i, \bot, \sigma_i) \quad \mathbf{obj}(\mathsf{X}, \bot, \rho_{\mathsf{X}})$$

A configuration is stuck, if it can not be reduced further but it is not terminated. We denote with $[\![\mathtt{e}]\!]_{\sigma,\rho}$ the evaluation of $\mathtt{e}$ with the stores σ and ρ. We write $\mathsf{C} \models \mathtt{e}$ iff $[\![\mathtt{e}]\!]_{\sigma,\rho} = \mathbf{tt}$ and the object whose store ρ is evaluated is understood.

We index the reduction relation with a tuple of active tasks. A singleton tuple (i) expresses that only i is active, a tuple (i, j) expresses that i is active and launches j. This allows us to reason about restricted behavior, i.e. $\mathsf{C} \Rightarrow_{(i)} \mathsf{C}'$ expresses that C' is reachable from C only by executing the task with id i.

Definition 3 (Run). *Let* $\mathsf{C}_1, \ldots, \mathsf{C}_n$ *be configurations. A run from* C_1 *to* C_n *is denoted* $\mathsf{C}_1 \Rightarrow \mathsf{C}_n$ *and defined as a tuple* $\mathsf{C}_1, \ldots, \mathsf{C}_n$ *with*

$$\mathsf{C}_1 \rightarrow_{I_1} \mathsf{C}_2 \rightarrow_{I_2} \ldots \rightarrow_{I_{n-1}} \mathsf{C}_n$$

for some tuples of task-ids $I_1, \ldots, I_{n-1}$*. We say that the run is annotated with* $I_1, \ldots, I_{n-1}$*. For simplicity, we assume that all runs are finite.*

Using the annotated tuples, we can define rooted runs: A run rooted in a task-id i is a run which only executes task i and tasks started by task i. Rooted runs allow one to reason about system behavior caused by a certain task.

Definition 4 (Rooted Runs). *Let the following be the graph of some tuple of tuples of task ids* $\mathcal{I} = (I_1, \ldots, I_n)$*:*

$$\mathcal{G}(\mathcal{I}) = (V, E)\ V = \{i \mid \textit{id } i \textit{ occurs in some } I_k\}\ E = \big\{(i,j) \mid \exists k < n.\ I_k = (i,j)\big\}$$

A tuple $\mathcal{I}$ *is* rooted in i*, if* $\mathcal{G}(\mathcal{I}')$ *is a tree with root* i *for each prefix* $\mathcal{I}'$ *of* $\mathcal{I}$*. A run rooted in* i*, denoted* $\mathsf{C}_1 \Rightarrow_i \mathsf{C}_n$*, is a run* $\mathsf{C}_1, \ldots, \mathsf{C}_n$ *annotated with* $\mathcal{I} = I_1, \ldots, I_{n-1}$*, such that* $\mathcal{I}$ *is rooted in* i*.*

Definition 5 ($\equiv^{\ominus}_{\mathsf{X}}$). *We write* $\mathsf{C} \equiv^{\ominus}_{\mathsf{X}} \mathsf{C}'$ *if two configurations are equal everywhere, except for the values of fields occurring in an expression* $\ominus$ *of object* X*:*

$$\mathsf{C} \equiv^{\ominus}_{\mathsf{X}} \mathsf{C}' \iff \exists \mathsf{C}''.\ \mathsf{C} = \mathsf{C}''\ \mathbf{obj}(\mathsf{X}, i, \rho) \wedge \mathsf{C}' = \mathsf{C}''\ \mathbf{obj}(\mathsf{X}, i, \rho') \wedge \bigwedge_{\mathtt{f} \notin \mathsf{fields}(\ominus)} \rho(\mathtt{f}) = \rho'(\mathtt{f})$$

3.1 Dynamic Logic

We use a dynamic logic, called SDL based on ABSDL [7] to reason about programs. SDL extends first-order logic with a modality for SYNC programs and allows us to reason about all possible runs of a method. We refer to [5, 7] for full formal details about ABSDL.

Definition 6 (Syntax). *Let* $\mathtt{v}$ *range over logical variables and* f *over function symbols. SDL-formulas* ϕ *and terms* t *are defined by the following syntax:*

$$\phi ::= \exists\ \mathtt{v}.\ \phi \mid \neg\phi \mid \phi \vee \phi \mid [\mathtt{s}]\phi \mid \langle\!\langle \mathtt{s} \rangle\!\rangle \phi \mid t \doteq t \qquad t ::= f(\overline{t}) \mid \mathtt{v}$$

The modality $[\mathtt{s}]\phi$ expresses that ϕ holds after the execution of $\mathtt{s}$ and at every suspension point within. We introduce $\langle\!\langle \mathtt{s} \rangle\!\rangle \phi$ below. A formula is valid if it holds in all models. The other formulas express constraints on given configurations. We assume a formalization of the heap with two function symbols store and select with the connecting axiom

$$\mathsf{select}(\mathsf{store}(\mathit{heap}, \circ, \mathtt{f}, \mathit{value}), \circ, \mathtt{f}) \doteq \mathit{value}$$

for every heap *heap*, object $\circ$, field $\mathtt{f}$ and value *value*. A modality-free formula holds in a configuration if the constraints are satisfied – select is interpreted such that $\mathsf{select}(\mathit{heap}, \circ, \mathtt{f}) \doteq \mathit{value}$) is satisfied in a configuration C if C has the form $\mathbf{obj}(\mathsf{X}, i, \rho)\ \mathsf{C}'$ and $\rho(\mathtt{f}) = [\![\mathit{value}]\!]_{\rho,\sigma} \wedge [\![\circ]\!]_{\rho,\sigma} = \mathsf{X}$ holds. The local store σ is also modeled globally, with one special function for each local variable. We assume for simplicity that all local variables have unique names.

Example 3. The following formula states that if in the beginning `o.f` is positive, then after the execution of `f = f+1;` in `o`, `o.f` is strictly positive.

$$\texttt{o.f} \geq 0 \rightarrow [\texttt{f = f+1;}]\texttt{o.f} > 0$$

The full semantics and a sequent calculus for validity are presented in [5, 7]. A sequent calculus operates on sequents of the form $\Gamma \Rightarrow \Delta$, where Γ, Δ are sets of SDL-formulas. Contrary to [7] we use the sequent calculus not to ensure that an invariant is preserved by a method, but only to check that the method establishes a certain post-condition at all suspension points.

We only show the rule for the **await** statement. The following rule is taken from [5, 7] and replaces the heap by a new function symbol to erase all knowledge. Afterwards, only the guard expression can be assumed. This mirrors the concurrency model, as other tasks may modify the heap. It also proves that the post-condition holds at each such point:

$$\frac{\Gamma \Rightarrow \phi, \Delta \qquad \Gamma \Rightarrow \{heap := newHeap\}\texttt{e} \rightarrow [\texttt{s}]\phi, \Delta}{\Gamma \Rightarrow [\textbf{await } \texttt{e; s}]\phi, \Delta} \ (\textbf{await})$$

We require a way to reason about all suspension points, except the first one. This is needed to verify that a method will fulfill a post-condition *after* being suspended at least once – it is not relevant whether the execution up to the first suspension satisfies the post-condition. Thus we use a special modality $\langle\!|\texttt{s}|\!\rangle\phi$ that expresses that ϕ holds after the execution of `s` and at each suspension point in `s`, except the first one. The calculus is the same, except that for the **await** statement, we use the following rule which does not check the post-condition. Note that afterwards the usual modality is used.

$$\frac{\Gamma \Rightarrow \{heap := newHeap\}\texttt{e} \rightarrow [\texttt{s}]\phi, \Delta}{\Gamma \Rightarrow \langle\!|\textbf{await } \texttt{e; s}|\!\rangle\phi, \Delta} \ (\textbf{await})$$

The connection to the language's SOS semantics follows from the correctness of the underlying validity calculus [7].

Lemma 1. *Let ϕ be a modality-free formula which contains function symbols only for the fields of object X, $\langle\!|\texttt{s}|\!\rangle\phi$ a formula and C_1 a configuration of the form*

$$\mathsf{C}_1 = \mathbf{tsk}(\mathsf{X}, i, \texttt{s}, \sigma)\ \mathbf{obj}(\mathsf{X}, i, \rho)\ \mathsf{C}'$$

If the proof for $\langle\!|\texttt{s}|\!\rangle\phi$ can be closed, then for every run $\mathsf{C}_1 \Rightarrow_i \mathsf{C}_n$ with intermediate configurations $\mathsf{C}_1, \mathsf{C}_2, \ldots, \mathsf{C}_n$ the following holds: At every position, except the very first, with a transition $\mathsf{C}_k \rightarrow_{(i)} \mathsf{C}_{k+1}$ such that i is active in X in C_k, but not in C_{k+1}, (i.e., these configurations execute suspension points) ϕ holds in C_k.

4 Dependencies for Condition Synchronization

A deadlock describes a stuck configuration, where tasks circularly depend on each other. To fix the notion of deadlock, we need to fix the notion of *dependency*.

Intuitively, a stuck task t depends on a task t' in a given configuration C, if the continued execution of t' leads to a configuration where t can continue its execution. If t' is stuck at some guard $\mathtt{b}$ too, then t depends on t' if the continuation of t' *in some configuration* C' *where* $\mathtt{b}$ *holds* leads to a configuration where t can continue its execution. We demand that C and C' are as similar as possible: they are equal everywhere but in the fields occurring in $\mathtt{b}$, as defined in Def. 5.

Definition 7. *We formally define a predicate* $\mathbf{dep}(\mathsf{C}, i, j)$ *which expresses that* i *depends on* j *in configuration* C*. The formalization is* not *in SDL but references SDL-formulas. To do so, we first define a family of predicates* n-$\mathbf{dep}(\mathsf{C}, i, j)$*, to model that* i *depends on* j *with at most* n *enforced continuations. Let* C *be a configuration of the form*

$$\mathbf{tsk}(\mathsf{X}, i, \textsf{await } \mathtt{e}; \mathtt{s}_i, \sigma_i) \quad \mathbf{tsk}(\mathsf{X}, j, \mathtt{s}_j, \sigma_j) \quad \mathbf{obj}(\mathsf{X}, \bot, \rho) \quad \mathsf{C}_0$$

The base predicate models that by executing only j*, a configuration can be reached, such that* $\mathtt{e}$ *evaluates to true:* 0-$\mathbf{dep}(\mathsf{C}, i, j) \equiv \mathsf{C} \not\models \mathtt{e} \wedge \exists \mathsf{C}'.\ (\mathsf{C} \Rightarrow_j \mathsf{C}' \wedge \mathsf{C}' \models \mathtt{e})$

The other predicates handle the case that both i *and* j *are blocked and* j *has the guard* $\mathtt{e}'$*: and by choosing a configuration* C'' *w.r.t.* C'*, the guard* $\mathtt{e}'$ *evaluates to true and in this configuration* i *depends on* j*.*

$$\begin{aligned} n\text{-}\mathbf{dep}(\mathsf{C}, i, j) \equiv \exists \mathtt{e}'.\ \exists \mathsf{C}', \mathsf{C}''.\ \mathtt{s}_j = \textsf{await } \mathtt{e}'; \mathtt{s}'_j \wedge \mathsf{C} \not\models \mathtt{e} \wedge \mathsf{C} \not\models \mathtt{e}' \wedge \mathsf{C} \equiv^{\mathtt{e}'}_{\mathsf{X}} \mathsf{C}' \\ \wedge\, \mathsf{C}' \models \mathtt{e}' \wedge \mathsf{C}' \not\models \mathtt{e} \wedge \mathsf{C}' \Rightarrow_j \mathsf{C}'' \wedge \big(\mathsf{C}'' \models \mathtt{e} \vee (n{-}1)\text{-}\mathbf{dep}(\mathsf{C}'', i, j)\big) \end{aligned}$$

Task i *depends on* j *in* C*, written* $\mathbf{dep}(\mathsf{C}, i, j)$*, if some* n-$\mathbf{dep}(\mathsf{C}, i, j)$ *holds.*

We can now distinguish between deadlock and starvation.

Definition 8 (Deadlock and Starvation). *The dependency graph of a configuration has its task ids as nodes and its dependencies as edges. A stuck configuration is* deadlocked *if its dependency graph contains a dependency cycle. A configuration is* starving*, if it is stuck, but not deadlocked.*

A starving configuration requires some condition $\mathtt{e}$ to become true, but no task can have such an effect. Sometimes an active process which tries to acquire a resource is also said to be starving, but in our framework this is abstracted to `await isAvailable(this.resource)` — all starving processes are stuck.

Example 4 (Deadlock and Starvation). Consider the program on the left in Figure 4. Its execution leads to the configuration

$$\mathbf{tsk}(\mathsf{X}, 1, \textsf{await } \texttt{f1; f2 = True;}, \sigma_1) \quad \mathbf{tsk}(\mathsf{X}, 2, \textsf{await } \texttt{f2; f1 = True;}, \sigma_2)$$
$$\mathbf{obj}(\mathsf{X}, \bot, \rho_{\mathsf{X}}) \quad \mathbf{tsk}(\mathsf{X}_0, 0, \bot, \sigma_0) \quad \mathbf{obj}(\mathsf{X}_0, \bot, \rho_{\mathsf{X}_0})$$

This configuration is deadlocked as for the dependency of task 1 on task 2 we can set $\mathsf{X}.\texttt{f1} = \mathsf{True}$ and for the dependency of task 2 on task 1 we can set $\mathsf{X}.\texttt{f2} = \mathsf{True}$. Now consider the right program is Figure 4. Its execution leads to

$$\begin{aligned} \mathsf{C} = &\mathbf{tsk}(\mathsf{X}, 1, \textsf{await } \texttt{f1; f2 = True;}, \sigma_1) \quad \mathbf{tsk}(\mathsf{X}, 2, \textsf{await } \texttt{f2; f1 = False;}, \sigma_2) \\ &\mathbf{obj}(\mathsf{X}, \bot, \rho_{\mathsf{X}}) \quad \mathbf{tsk}(\mathsf{X}_0, 0, \bot, \sigma_0) \quad \mathbf{obj}(\mathsf{X}_0, \bot, \rho_{\mathsf{X}_0}) \end{aligned}$$

C is starving, as task 1 does *not* depend on task 2: no configuration can be chosen to continue task 2, so it leads to a configuration that evaluates $X_1.f1$ to True.

```
object X{
  Bool f1 = False; Bool f2 =
 False;
  m(){ await f1; f2 = True; }
  n(){ await f2; f1 = True; }
}
main{async X.m(); async X.n();}
```

```
object X{
  Bool f1 = False; Bool f2 =
 False;
   m(){ await f1; f2 = True; }
   n(){ await f2; f1 = False; }
}
main{async X.m(); async X.n();}
```

Fig. 4. Two example programs: The left will deadlock, the right will starve.

It is undecidable in general whether a configuration is deadlocked, as the computation of the dependency includes the computation of all effects caused by the program following a guard. Program and guard are both turing-complete, thus one can define a function encoding the universal turing machine and check in the guard for some property of the output of another turing machine, which is computed/encoded in the code of another method.

Proposition 1. *Given a stuck configuration* C*, it is not decidable whether* C *is deadlocked or starving.*

Indeed even the dependency relation is undecidable. This result may appear discouraging, but the presented notion of deadlock captures *the intent of the designer* more precisely than notions which do no take the information flow through the heap into account and do not differentiate between deadlock and starvation. The aim of our analysis is to present clues to the designer where the intended control flow has circular dependencies. It does not aim to catch *any* kind of error and is not supposed to catch implementation bugs, where every erroneous state is undesirable. The aim is to catch specific *logical* errors in the design of the control flow. Under these assumptions, undecidability is not a deal-breaker. Indeed, if the notion would be decidable, it would restrict the possible guards – our aim however is to give the designer full freedom and support him with clues where it might deadlock, not guarantee complete error-freedom.

Similarly, it is useful to distinguish between deadlock and starvation. Both notions describe erroneous states, but the reason are different design flaws. Also, starvation is not always undesirable. Consider the following method:

```
server(){
  await requestList != Nil;
  //handle requests
  async this.server();
}
```

Here, the object buffers and handles multiple requests at once. This pattern

is used in practice [15]. Starvation is only caused by a lack of requests, not erroneous control flow. Similarly, the right code in Figure 1 will terminate in a starving configuration, as `m1` does not depend on `m2`. A starvation analysis would also be useful, but is out of the scope for this work.

5 Analyzing Condition Synchronization

To detect deadlocks, the *abstract dependency graph* is computed. The abstract dependency graph subsumes all dependency graphs of reachable stuck configurations in a program: If the dependency graph of a reachable stuck configuration has a circular dependency, then the abstract dependency graph also has one.

Our approach extends the one of Flores-Montoya et al. [8] and the implementation thus handles a language with condition synchronization and synchronization on futures, i.e., termination of tasks. For presentation's sake, we only define the object-insensitive abstract dependency graph. Improvements of [8] can still be applied, e.g., their main improvement relies on a *may-happen-in-parallel* analysis, which is extended for condition synchronization in [2]. The abstract dependency graph is defined syntactically. Let P be a program.

Definition 9. *Let* $\mathsf{X}_1, \ldots, \mathsf{X}_n$ *be all objects in* P *and* $\mathtt{m}_{i,1}, \ldots, \mathtt{m}_{i,o}$ *the methods of* X_i*. The* abstract dependency graph $\mathbb{A}(\mathsf{P}) = (V, E)$ *is defined as follows:*

- *The nodes are all methods, i.e.* $V = \left(\mathtt{m}_{i,j}\right)_{\substack{i \leq n \\ j \leq o}}$
- *Edges connect methods with writes into a field with methods which synchronize on this field:* $(\mathtt{m}_{i,j}, \mathtt{m}_{k,l}) \in E$ *iff there is a field* `f` *such that* $\mathtt{m}_{i,j}$ *contains a guard with* `f` *and* $\mathtt{m}_{k,l}$ *contains* `f = e` *or a call to a method doing so.*

Note that a guard may contain multiple fields and that methods on different objects may depend on each other. At this point, we do not analyse here whether call or write statement are in a branch or in dead code.

Example 5. Consider the following code and its abstract dependency graph

```
1 object X {
2  Bool b1 = False;  Bool b2 = False;
3  m(){ b1 = True; await b2; }
4  n(){ await !b1;  b2 = True; }
5 }
6 main { async X.m(); async X.n();}
```

To incorporate the side-effects of computations, we make two additional steps. The first improvement aims to discards cycles because there is no *reachable* deadlock configuration to which they correspond. In Definition 9, the whole method was checked for written fields. A cycle, however, only represents some concrete configuration where the processes hold *at specific guards*: every field in a deadlocked configuration must be written *after* some synchronization statement.

Definition 10 (Feasibility). *A cycle* $\mathtt{m}_1, \ldots, \mathtt{m}_n$ *in* $\mathbb{A}(\mathsf{P})$ *is* feasible*, if for each* $k < n$*, every write causing the edge* $(\mathtt{m}_k, \mathtt{m}_{k+1})$ *is after the first guard of* $\mathtt{m}_{k+1}$*.*

Nonfeasible cycles contain edges that refer to information flow that happens during the execution of an involved method, but *before* the stuck configuration is reached:

Example 6. Consider again Example 5. The edge from `X.n` to `X.m` is added, because the field `b1` is written in `m` and read in a guard in `n`. This edge is missing in all concrete dependency graphs of reachable stuck configurations, because in the stuck configurations `X.m` has already reached its guard and thus *will not change* `b1`. I.e., the cycle (`X.n`,`X.m`,`X.n`) is not feasible.

We may increase the accuracy further by analyzing the transmitted information: We ensure that every edge is refering to a write statement which actually *may release the guard*. To do so for a guard `e` and a method $\mathtt{m}_{k,l}$ with method body `s`, we must ensure that after some suspension of inside of `s` the guard `e` evaluates to true. We may ignore the first suspension, as all side effects before it cannot influence the heap afterwards. I.e., if the formula $\langle\!|\mathtt{s}|\!\rangle\neg\mathtt{e}$ is valid, then after no execution of $\mathtt{m}_{k,l}$ can resolve the blocking guard and we can remove the dependency edge.

Definition 11 (Refined Abstract Dependency Graphs). *Let $\mathfrak{G} = (V, E)$ be an abstract dependency graph. Let $(\mathtt{m}_{i,j}, \mathtt{m}_{k,l})$ be an edge, added because of a statement* **await** `e` *in $\mathtt{m}_{i,j}$. Let* `s` *be the body of $\mathtt{m}_{k,l}$ The edge $(\mathtt{m}_{i,j}, \mathtt{m}_{k,l})$ is* dispensable *if the formula $\langle\!|\mathtt{s}|\!\rangle\neg\mathtt{e}$ holds.*

The refined abstract dependency graph *of a program is the graph that results from removing all dispensable edges from its abstract dependency graph.*

The use of the $\langle\!|\cdot|\!\rangle$ is necessary, as we only reason about *stuck configurations*, thus we can ignore any side effects that happen before the first guard - they do not refer to information flow that may release another guard afterwards.

Example 7. Consider the right program in Figure 4. As discussed this program will starve, but deadlock. The left graph below is its abstract dependency graph, the right graph the refined abstract dependency graph:

Theorem 1 (Soundness). *If a program has a reachable deadlocked configuration, then its refined abstract dependency graph has a feasible cycle.*

6 Evaluation

We implemented our approach in the SACO [1] framework for coreABS and use KeY-ABS [5] as the theorem prover to check the condition for refined abstract dependency graphs[1]. Existing tools did not support conditional synchronization, so only six case studies made use of this feature. Our work is the first to implement a deadlock analysis for boolean guards and we rely on micro-benchmarks

```
1  class Server implements S{
2   List<Work> wList = Nil;
3   Int status = 0;
4   Unit in_pool(){
5    await status == 1;
6    wList = 1;}
7   Unit add_worker(Fut f){
8    await f?;
9    wList = [new Work()| wList];}
10  Unit run(){
11   this!init_all();
12   Fut f = this!in_pool();
13   this!add_worker(f) }
14  Unit init_all(){
15   status = 1;
16   await length(wList) >= 2;
17   wList = [new Work()| wList];
18   status = 2; }}
```

Fig. 5. An ABS class modeling the internal synchronization structure of a server during initialization. The main block is omitted and the ABS code is prettified.

to evaluate on a wider code base. Case studies and micro-benchmarks cover full coreABS, including loops. The implementation is fully automatic.

We can show that the right example in Figure 4 is deadlock free. Figure 5 mixes conditional synchronization and synchronization on futures. The implementation can deal with deadlocks where some dependencies are caused by futures and some dependencies are caused by condition synchronization. This example requires the application of the theorem prover: two false positive deadlock risks are found otherwise. Similarly, Example 2 requires the theorem prover to be shown as deadlock-free.

We analyzed the two largest examples in ABS, the FredHopper trading and replicate systems which model industrial software systems [6], and found 20 deadlock risks in the trading system and 52 in the replicate system. One reason for this is that the replicate system uses deployment components [13] modeling cloud architecture, which are not supported by KeY-ABS. In [2, 8] the trading system was already analyzed in a setting with conditional synchronization as deadlock free. In that work, only the MHP analysis was adjusted for conditional synchronization, the deadlock analysis however was not sound and does not detect the deadlock in the left program in Figure 4. We were able to manually identify all 20 deadlock risks as false postives and confirm that the trading system is deadlock free. Manual post-processing is acceptable as the tool outputs the methods involved in the deadlock risk and 18 of the 20 deadlock share one edge.

We analyzed the non-trivial models for industrial architecture from HyVar [18] and FormbaR [14]. Additionally, we evaluated an ABS model for weak memory [15], an ABS model for resource consumption in YARN clusters [17], and the Compugene model for computational biology[2]. The analysis returns 3 (resp. 3 and 1) potential deadlocks, which are easily manually identified as false positives. The false positives in the YARN model are again due to the use of deployment components. The analysis confirms deadlock-freedom of an ABS Blockchain model [19]. The right side of Table 1 summarizes our evaluation on these case studies. The critical edge column shows how many edges needs to be

[1] Available under formbar.raillab.de/deadlock
[2] http://www.compugene.tu-darmstadt.de

Selected Microbenchmarks (5 of 42)				
Name	LoC	deadlock-free	time	found deadlocks
back_dead	39	×	8ms	1
OneQueue	37	×	13ms	2
Figure 5	43	✓	7ms	0
Transitive	52	×	10ms	1
Loop	39	×	8ms	1
Case Studies				
Study	LoC	potential deadlocks	time	critical edges
BlockChain	620	0	1312ms	-
Compugene	860	1	83ms	1
Memory	351	3	49ms	2
YARN	199	3	144ms	3
HyVar	632	6	200ms	2
trading	1466	20	31s	3
replicate	2101	52	5s	11
FormbaR	2200	Timeout		

Table 1. Evaluation on selected examples

removed from the graph to remove all cycles. The lower the number, the more feasible manual post-processing is. Except the mentioned industrial examples, the Compugene, HyVar and weak memory models, all examples are written by us. The tool was run on a Intel E5-2643 with 6 cores 3,4 GHz and 64 GB RAM.

Our analysis does not scale only for the FormbaR model. This has two main reasons: (1) FormbaR models communication during railway operations and contains little computation, while the other example systems are less communication-heavy. (2) FormbaR makes heavy use of maps and contains several guards that read 4 fields of the class and every field is written in several methods - the model contains a lot of information flow through fields. Maps complicate analysis, as they require to ensure that the keys are passed around correctly. Such global properties cannot be derived by analyzing methods in isolation.

7 Conclusion

We presented an approach for deadlock detection in presence of conditional synchronization, which integrates a theorem prover to analyze side-effects. The implementation is the first sound deadlock analysis for full coreABS. We are able to analyze all ABS case studies, but are not precise if models contain synchronization points that access many fields of a class: the abstract dependency graph subsumes all information flow in a program and is highly connected in those cases. This reflects the inherent difficulty of reasoning about arbitrary side-effects.

Discussion of the Use of Deductive Verification Our analysis integrates a heavy-weight deductive verification tool into a light-weight static analysis. This allows us to reason about heap memory beyond analyzing the field names occurring in a method, but also offers other beneficial features from a design perspective.

Theorem provers have a clear interface and our implementation is not monolithic: Our deadlock tool benefits from any future advance in the precision or performance of KeY-ABS. Every invocation of KeY-ABS is caused by a pair of

one guard and one method. This gives us modularity of the analysis results: If method and guard are unchanged, the prover does not have to be run again. We are able to handle unbounded data types and recursion without performance loss: KeY-ABS works on symbolic values and analyzes single methods. Non-termination is handled implicitly and we do not need to provide a maximal number of unrolling for loops or similar. Contrary to that, model checking would involve rerunning the whole program after each change and relies on finite domains and traces. We are still fully automatic, but we propose that in some situations it would be acceptable to interact with the theorem prover.

Related Work To the author's best knowledge, no deadlock analysis for condition synchronization in a object-oriented setting has been proposed. Some work has been done on simpler concurrency models, e.g., Owicki and Gries [21], which does not to models with an arbitrary number of threads. For Active Objects (without condition synchronization) the following approaches are proposed: (1) The discussed approach of Flores-Montoya et al. [8]. (2) Giachino et al. [9] use behavioral types to detect deadlocks in ABS code. *Contracts*, descriptions of the dependency-structure of methods, are inferred and cycles are detected in their composition. For boolean guards, manual annotations are proposed, but not implementeted and no inference algorithm is given. (3) Gkolfi et al. [10] use Petri Nets for deadlock detection and do not consider or discuss boolean guards.

As described, conditional synchronization is similar to *condition variables*. Leino et al. [16] presented an approach to deadlock detection of locks that generalizes to condition variables. Deadlocks are checked on a manually annotated global order for releasing and acquiring locks, receiving and sending messages over channels, and joining on threads. de Carvalho-Gomes et al. [4] translate Java programs into colored Petri nets for deadlock detection. While translation into Petri nets and the analysis of these Petri nets are automatic, the approach requires manual annotations. Recently, Hamin and Jacobs [11] presented an approach that works directly on condition variables in C code, based on symbolic execution and verified in Coq. Java PathFinder [22] also uses symbolic execution, but does so on low-level primitives in Java bytecode.

Future Work For precision, we aim to integrate user-provided specifications in SDL– while this would no longer be fully automatic, such invariants are available for many ABS case studies. To automate rejection of assumed false positives, we plan to adopt the approach of Albert et al. [3] to generate tests. For scalability, we plan an incremental approach to summarize detected cycles based on critical edges. We did not discuss starvation analysis, which is also an open question.

Acknowledgments

This work is supported by FormbaR, part of AG Signalling/DB RailLab. We thank Antonio Flores-Montoya and Michael Lienhardt for insightful discussions and the anonymous reviewers for valueable feedback.

References

1. E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: static analyzer for concurrent objects. In *TACAS 2014, Proceedings*, 2014.
2. E. Albert, A. Flores-Montoya, and S. Genaim. May-happen-in-parallel analysis with condition synchronization. In *Foundational and Practical Aspects of Resource Analysis (FOPARA 2015)*, 2015.
3. E. Albert, M. Gómez-Zamalloa, and M. Isabel. Combining static analysis and testing for deadlock detection. In *iFM 2016, Proceedings*, pages 409–424, 2016.
4. P. de Carvalho Gomes, D. Gurov, and M. Huisman. Specification and verification of synchronization with condition variables. In *FTSCS*, volume 694 of *Communications in Computer and Information Science*, pages 3–19, 2016.
5. C. C. Din, R. Bubel, and R. Hähnle. KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In A. P. Felty and A. Middeldorp, editors, *Automated Deduction - CADE'15, Proceedings*, pages 517–526, 2015.
6. C. C. Din, R. Bubel, R. Hähnle, E. Giachino, C. Laneve, and M. Lienhardt. Deliverable D3.2 Verification of project FP7-610582 (ENVISAGE), Mar. 2015. available at http://www.envisage-project.eu.
7. C. C. Din and O. Owe. Compositional reasoning about active objects with shared futures. *Formal Asp. Comput.*, 27(3):551–572, 2015.
8. A. Flores-Montoya, E. Albert, and S. Genaim. May-happen-in-parallel based deadlock analysis for concurrent objects. In *FORTE 2013, Proceedings*, 2013.
9. E. Giachino, C. Laneve, and M. Lienhardt. A framework for deadlock detection in coreABS. *Software & Systems Modeling*, pages 1–36, 2015.
10. A. Gkolfi, C. C. Din, E. B. Johnsen, M. Steffen, and I. C. Yu. Translating active objects into colored Petri nets for communication analysis. In *FSEN*, volume 10522 of *LNCS*. Springer, 2017.
11. J. Hamin and B. Jacobs. Deadlock-free monitors. In *ESOP*, volume 10801 of *Lecture Notes in Computer Science*, pages 415–441. Springer, 2018.
12. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *FMCO*, pages 142–164, 2010.
13. E. B. Johnsen, R. Schlatte, and S. L. T. Tarifa. Modeling resource-aware virtualized applications for the cloud in real-time ABS. In T. Aoki and K. Taguchi, editors, *ICFEM 2012, Proceedings*, pages 71–86, 2012.
14. E. Kamburjan and R. Hähnle. Uniform modeling of railway operations. In C. Artho and P. Ölveczky, editors, *Proc. Fifth Intl. Workshop on Formal Techniques for Safety-Critical Systems (FTSCS)*, volume 694 of *CCIS*. Springer, 2016.
15. E. Kamburjan and R. Hähnle. Prototyping formal system models with active objects. In *ICE'18, to appear*, 2018. Preprint: http://formbar.raillab.de/en/paper/.
16. K. R. M. Leino, P. Müller, and J. Smans. Deadlock-free channels and locks. In *ESOP'10, Proc.*, pages 407–426, 2010.
17. J. Lin, I. C. Yu, E. B. Johnsen, and M. Lee. ABS-YARN: A formal framework for modeling hadoop YARN clusters. In *FASE*, volume 9633 of *Lecture Notes in Computer Science*, pages 49–65. Springer, 2016.
18. J.-C. Lin, J. Mauro, T. B. Røst, and I. C. Yu. A Model-Based Scalability Optimization Methodology for Cloud Applications. In *7th IEEE International Symposium on Cloud and Service Computing, IEEE SC2 2017*, 2017.
19. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

20. N. Ng and N. Yoshida. Static deadlock detection for concurrent go by global session graph synthesis. In *CC*, pages 174–184. ACM, 2016.
21. S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6:319–340, 1976.
22. C. S. Pasareanu, W. Visser, D. H. Bushnell, J. Geldenhuys, P. C. Mehlitz, and N. Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Autom. Softw. Eng.*, 20(3):391–425, 2013.