I can see clearly now: Clairvoyant Assertions for Deadlock Checking*

Ole Jørgen Abusdal[1], Crystal Chang Din[2], Violet Ka I Pun[1], and Volker Stolz[1]

[1] Western Norway University of Applied Sciences, Norway
{ojab,vpu,vsto}@hvl.no
[2] University of Bergen, Norway
Crystal.Din@uib.no

Abstract. Static analysers are traditionally used to check various correctness properties of software. In the face of refactorings that can have adverse effects on correctness, developers need to analyse the code after refactoring and possibly revert their changes. Here, we take a different approach: we capture the effect of the Hide Delegate refactoring on programs in the ABS modelling language in terms of the base program, which allows us to predict the correctness of the refactored program. In particular, we focus on deadlock-detection. The actual check is encoded with the help of an additional data structure and assertions. Developers can then attempt to discharge assertions as vacuous with the help of a theorem prover such as KeY. On the one hand, this means that we do not require a specific static analyser nor theorem prover, but rather profit from the strength and advances of modern tool support. On the other hand, developers can choose to rely on existing tests to confirm that no assertion is triggered before executing the actual refactoring. Finally, we argue the correctness of our over-approximation.

Keywords: refactoring, deadlock detection, active object languages

1 Introduction

Refactoring is an important software engineering activity. Current tool support in IDEs provides a broad selection of well-known refactorings. These refactorings give no guarantees as to the expected behaviour and have been known to err in this regard in the past, see [21], beyond hopefully still producing compilable code afterwards. We follow Fowler's stipulation that refactorings should preserve behaviour. This is already difficult to check before executing a refactoring at the best of times, and complete support for proving needs sophisticated frameworks such as KeY [1,2,22].

In earlier work, we have introduced assertions during refactoring [9]. These assertions capture the correctness conditions for a refactoring, and place a lighter burden on the developers, in that they do not have to provide proofs in unfamiliar, advanced, incomplete or even non-existing frameworks, but can use their

* Partially supported by DIKU/CAPES project "Modern Refactoring".

tools of the trade such as tests and coverage analysis to judge whether the refactored system has been sufficiently exercised to confirm expected behaviour.

These assertions could of course in principle be discharged with program provers. In this paper, we make two contributions: first, we focus on a novel domain of refactoring for active object programs; specifically ABS [14,16][3] programs, where a direct application of well-known object-oriented refactorings potentially leads to surprising results such as deadlocks [23], and second, we present an approach where we insert assertions encoding the correctness conditions in the code *before* refactoring, such that applicability of a refactoring can be checked either *dynamically* (through testing) or *statically* (through proving).

Although, in general, proofs for these conditions can be quite involved in non-trivial settings, such as in programs with unbounded concurrency, it is our standpoint that for easy scenarios, e.g., for a statically known number of objects with a fixed communication topology, the proof-support should be sufficient.

As a proof of concept, we show how to derive the required assertions for the Hide Delegate refactoring. We are motivated by a belief that for the above class of programs we can make use of automated discharging of the assertions (or counter example derivation). For more involved programs, this should at least narrow down the scope for further investigation and guide developers to cases they have to consider before concluding that the refactoring will be correct.

A refactoring is correct and can then safely be applied if all assertions can be discharged. This approach also has the advantage that any remaining assertions will be refactored together with the program, should the developers choose to proceed with applying the refactoring. The assertions then, in the spirit of our earlier results, still serve as runtime checks: a passing assertion indicates that the subsequent synchronisation will not deadlock.

The KeY system [1] has been developed for over two decades. It started in 1998 by Hähnle et al. at Karlsruhe Institute of Technology. The original KeY system supports verification of sequential Java programs. A new version of the KeY system, i.e., KeY-ABS [7], was introduced in 2015. KeY-ABS supports symbolic execution, assertion checking and verification of history-based class invariants for concurrent ABS programs. In this paper, we present a deadlock detection framework for ABS and discuss why KeY-ABS is a suitable tool to implement this analysis approach. We also provide directions on where further effort might be a good investment in the KeY-ABS system.

The remainder of the paper is structured as follows: Section 2 briefly introduces the ABS language and how deadlocks can occur. Section 3 describes an assertion transformation to detect deadlocks, and Section 4 presents the approach to deadlock detection for *to-be-refactored* programs *before* refactoring. Section 5 discusses how we can use KeY-ABS to reason about the transformed program. Finally, we explore the related work in Section 6, and conclude the paper with a discussion on some limitations and future work in Section 7. The example presented in this paper is available as a git repository at `https://github.com/selabhvl/stolz-srh60-artefact`.

[3] https://abs-models.org/

$$
\begin{array}{rcl}
cn & ::= & \epsilon \mid fut \mid object \mid invoc \mid cog \mid cn\ cn \\
fut & ::= & fut(f, val) \\
object & ::= & ob(o, a, p, q) \\
q & ::= & \epsilon \mid p \mid q\ q \\
invoc & ::= & invoc(o, f, m, \overline{v}) \\
s & ::= & s; s \mid x = rhs \mid \textsf{suspend} \mid \textsf{await}\ g \mid \textsf{skip} \\
 & & \mid \textsf{if}\ b\ \{s\}\ [\ \textsf{else}\ \{s\}\ ] \mid \textsf{while}\ b\ \{s\} \mid \textsf{return}\ e \mid \textsf{cont}(f) \\
rhs & ::= & e \mid \textsf{new}\ [\textsf{local}]\ C[(\overline{e})] \mid e!m(\overline{e}) \mid e.m(\overline{e}) \mid x.\textsf{get}
\end{array}
\qquad
\begin{array}{rcl}
cog & ::= & cog(c, act) \\
val & ::= & v \mid \bot \\
a & ::= & T\ x\ v \mid a,\ a \\
p & ::= & \{l \mid s\} \mid \text{idle} \\
v & ::= & o \mid f \mid b \mid t \\
act & ::= & o \mid \varepsilon
\end{array}
$$

Fig. 1: Runtime syntax of ABS, o, f, c are identifiers of object, future, and cog

2 The ABS language and deadlocks

In this section, we briefly introduce the ABS language [16], the active object language that the work is based on. We will first present the runtime syntax and then we show how deadlocks can be introduced by code refactoring in the language.

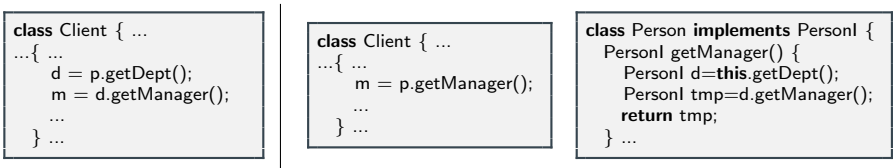
2.1 The ABS language

ABS is a modeling language for designing, verifying, and executing concurrent software. It has a Java-like syntax and actor-based concurrency model [15], which uses *cooperative scheduling* of method activations to explicitly control the internal interleaving of activities inside a concurrent object group (*cog*). A cog can be conceptually seen as a processor containing a set of objects. An object may have a set of processes, triggered by method invocations, to be executed. Inside a cog, at most one process is *active* while the others are *suspended* in the process pool of the corresponding objects.

Process scheduling is non-deterministic, but is explicitly controlled by the *processor release points* in the language. Such a cooperative scheduling ensures data-race freedom inside a cog. In addition, objects are hidden behind interfaces and all fields are private to an object. Any non-local read or write to fields must be performed explicitly through method invocations. Different cogs can only communicate through asynchronous method calls. Note that a synchronous method call to objects on a different cog will be translated to an asynchronous one that is immediately followed by a blocking **get** operation. Thus, the cog in which the caller resides will be blocked until the method returns. In contrast, synchronous calls within the same cog will only lead to transferring the control of the cog from the caller to the callee, i.e., no cog will be blocked.

The runtime syntax of ABS is presented in Fig. 1[4], where overlined terms represent a (possibly empty) lists over the corresponding terms, and square brackets [ ] denote optional elements. A configuration *cn* either is empty or consists

[4] We have adopted the new version of the syntax for object creation instead of the one presented in [16].



(a) Before (b) After

Fig. 2: Before/after Hide Delegate

of futures, objects, invocation messages and cogs. A *future* $fut(f, v)$ has an identifier f and a value v ($\perp$ if the associated method call has not returned). An object is a term $ob(o, a, p, q)$ consisting the object's identifier o, a substitution a representing the object's fields, an active process p and a pool of suspended processes q, where a substitution is a mapping from variable names to values. A process p is idle or consists of a substitution of local variables l and a sequence of statements s, denoted as $\{l \mid s\}$. Most of the statements are standard. The statement **suspend** unconditionally suspends the active process and releases the processor, while the statement **await** g suspends the active process depending on the guard g, which is either Boolean conditions b or return tests $x?$ that evaluate to true if the value of the future variable x can be retrieved; otherwise false. The statement **cont**(f) controls scheduling when local synchronous calls complete their execution, returning control to the caller.

Right-hand side expressions *rhs* for assignments include object creation within the current cog, denoted as **new local** $C(\overline{e})$, and in a new cog, denoted as **new cog** $C(\overline{e})$, asynchronous and synchronous method calls, and (pure) expressions e. An invocation message $invoc(o, f, m, \overline{v})$ consists of the callee o, the future f to which the call returns its result, the method name m, and the actual parameter values $\overline{v}$ of the call. Values are object and future identifiers, Boolean values, and ground terms from the functional subset of the language. For simplicity, classes are not represented explicitly in the semantics, as they may be seen as static tables.

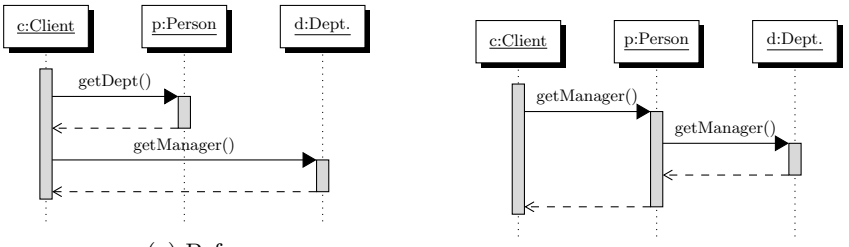
We do not further detail the syntax and semantics of ABS in this paper, but refer the readers to [16] for the complete details.

2.2 Deadlocks introduced by refactoring

Fig. 2 presents snippets of ABS code before and after a Hide Delegate refactoring that may introduce deadlocks in an actor setting, which is described in [23].

The effect of introducing deadlocks by this refactoring can be summarised by inspecting the difference between the two sequence diagrams in Fig. 3 showing how synchronous calls change, and by considering the possible assignment of objects to cogs shown in Fig. 4.

Fig. 3a shows that a Client is first communicating with Person, then with Dept, while Fig. 3b shows that the Client in the refactored program delegates invoking getManager() to Person. Assume that we have a set of at least three objects



(a) Before (b) After

Fig. 3: Effect of Hide Delegate refactoring

$\{c, d, p, \ldots\}$ all placed in some cogs. We represent this information by a mapping of object identifiers to cog identifiers. A placement that is without deadlocks before, but with a deadlock after refactoring is $\{c \mapsto 1, d \mapsto 1, p \mapsto 2, \ldots\}$, i.e., objects c and d reside in a cog with identifier 1, while object p is located in a cog with identifier 2.

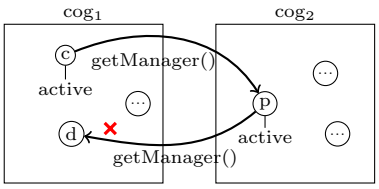


Fig. 4: A deadlock

Fig. 4 depicts this placement, under which these three objects can be deadlocked. We observe that object c is blocking cog_1 while it is waiting for object p in cog_2 to complete processing getManager(), where object p in turn invokes getManager() on object d in cog_1. Since cog_1 is blocked by object c, object d will not be able to execute the method. Consequently, object p will never finish executing getManager().

2.3 A wait-for relation between cogs

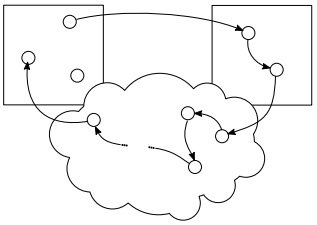


Fig. 5: A deadlocking call chain

Consider the arbitrary call chain shown in Fig. 5 and imagine traversing through the execution path resulting in this chain. Although we cannot yet determine if there exists a deadlock after the first call in the chain, we know that a synchronous call to an object on another cog will block the cog of the caller, i.e., no other object residing in the same cog can proceed. The cog of the caller will remain blocked until the called method returns. After the first call in the chain, we say that the caller cog and the callee cog are in a *wait-for* relation, i.e., the caller cog is waiting for the callee cog. We generalize the *wait-for* relation to also any blocking operation including the waiting for futures to be resolved. Thus, if an object requests the value of a future using the blocking **get** operation, we say its cog and the cog in which the future will be resolved are in a *wait-for* relation. This may be an over-approximation in the case where an **await** statement precedes the **get** operation.

Wait-for relation and deadlocks. For each configuration of a given ABS program, we can derive the current wait-for relation indicating if a cog is waiting for another one. A cycle in this relation indicates that there exists a deadlock involving the cogs that form the cycle. Any detection of a cycle can be done prior to any program points that contain a blocking or possibly blocking operation, which are:

- Blocking: Synchronous method calls $\mathsf{x} = \mathsf{o.f}(\bar{e})$ where the caller **this** and the callee o reside in different cogs.
- Possibly blocking: At any statement $\mathsf{x} = \mathsf{f.}$**get** irrespective of whether the caller **this** and the future f are in different cogs.

Note that although synchronous calls within the same cog in ABS do not lead to deadlocks, an asynchronous call to an object o residing in the same cog as the caller may lead to a deadlock, e.g., **Fut**<Unit> f = o!m(); x = f.**get**. Our analysis will correctly detect this deadlock prior to the **get** operation. However, in the case where an **await** statement precedes the **get** operation, e.g., **Fut**<Unit> f = o!m(); **await** f; x = f.**get**, our analysis will give rise to a *false positive* (see the next section for the details).

3 Program transformation for deadlock checking

In this section, we introduce a general transformation mechanism to inject assertions into the program to detect deadlocks at runtime based on the *wait-for* relation.

3.1 Assertion transformation

To perform deadlock checking on a program based on the wait-for relation in the form of runtime assertions, the relation needs to be updated along any possible call chain. Such an update requires information about the cog placement of each object, which is not explicitly available in an ABS program, but can be inferred by slightly transforming the program. Fig. 6 captures such a transformation, which enables the construction of the wait-for relation in the form of a data structure (w4) and subsequently the detection of deadlocks. In the figure, we have taken some liberties for a denser presentation. We use a pseudo-syntax, e.g., **class** $\mathsf{C}(\overline{\mathsf{T}\ \mathsf{e}})\_\{\_\{\_\}\_\}$ refers to a pattern matching any class where C corresponds to a class name. We explain in the following how the transformation is performed.

Any object creation performed through **new** will place the object in a new cog, whereas **new local** will place the object in the same cog as the one executing the constructor call. Thus, to associate every object with a cog, we modify every constructor declaration, **class** $\mathsf{C}(\overline{\mathsf{T}\ \mathsf{e}})\_$, such that it is parametrised with a cog identifier and a map that links object references to cog identifiers i.e., **class** $\mathsf{C}(\mathsf{CogId\ id}, \mathsf{CogMap\ cogs}, \overline{\mathsf{T}\ \mathsf{e}})\_$, as shown in Fig. 6a. Additionally, in the init block of each class, we inject code to update the cog map to link any class

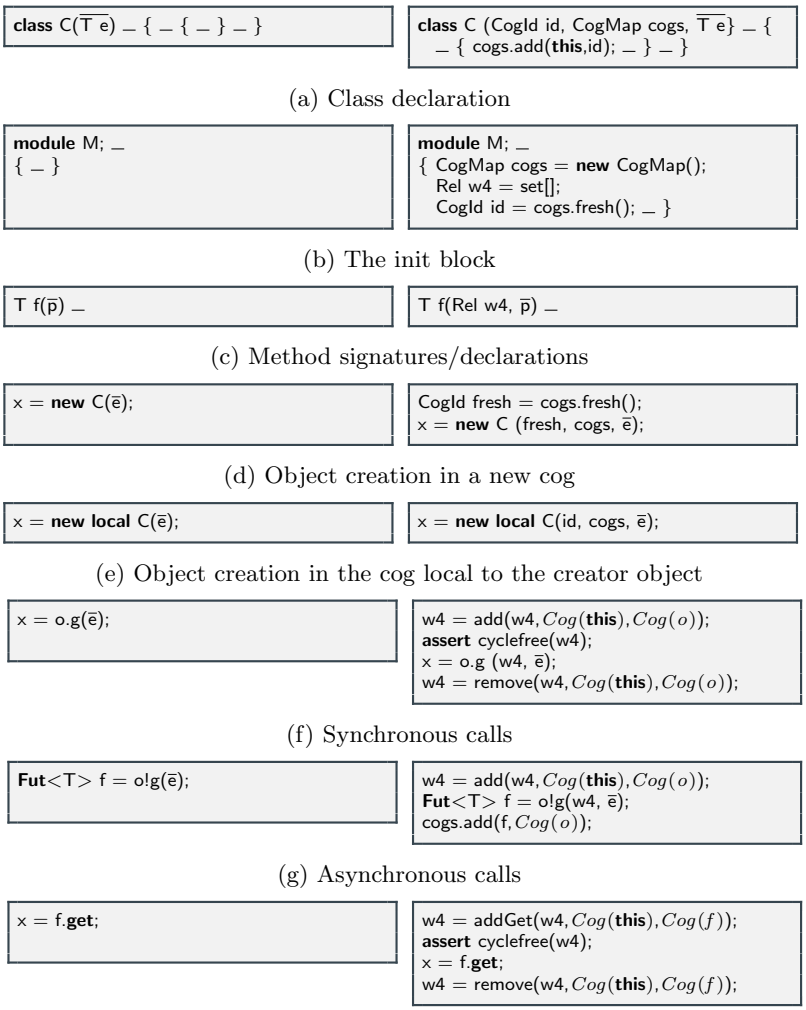


Fig. 6: The assertion transformation, the notation _ is a wildcard match for the expected syntactic entity at its position.

instance to the cog identifier it receives as constructor parameter, as shown in Fig. 6b, where cogs.fresh() is a function returning a fresh cog identity. Note that the cog map is one single object in the program that all other objects, or scopes in the case of the program main block, has a reference to. As such it can dispense of the freshness requirement through also being able to emit fresh identifiers. An empty wait-for relation is also created in the init block of the program, where w4 a functional data structure capturing the wait-for relation on cog identifiers, which can be a set containing pairs of cog identifiers.

Naturally, we must modify every constructor invocation to reflect our change to constructors (see Figs. 6d and 6e). We either record a *fresh* cog identifier for the case of a constructor invocation starting with **new**, or the identifier of the cog where the object invokes **new local**. With our change to the constructor parameters of all classes, we ensure that there is a reference to the cog map in every scope for any updates to the wait-for relation (w4).

The signature of all method definitions is transformed to receive w4 as one of the parameter (see Fig. 6c). Correspondingly, all method invocations are transformed to receive the current value of w4 as the first parameter, as shown in Figs. 6f and 6g Statements are also added to update the wait-for relation. Fig. 6f presents the transformation for *synchronous calls*. We first add the call chain information, represented as a pair of cog identifier $\langle Cog(\textbf{this}), Cog(o)\rangle$, to w4 before the synchronous call to object o is invoked, where the function $Cog(o)$ returns the identifier of the cog in which the object o is residing. This pair is removed after the call is made and returns. The update mechanism maintains an *irreflexive* invariant for the wait-for relation for synchronous calls by never adding a pair where $Cog(o_1) = Cog(o_2)$. The wait-for relation is handled similarly for *asynchronous calls*, as shown in Fig. 6g. For each statement **Fut**<T> f = o!g($\bar{e}$), we register the future variable f in the cog map with the function cogs.add(f, $Cog(o)$), such that the **get** rule (see Fig. 6h) can later retrieve this information. Note that although the call chain information is added to w4 prior to the method invocation, this information is *not* removed because it is unclear when the call returns.

On any retrieval of values in futures, i.e., f.**get**, in Fig. 6h, we first update w4 with addGet to indicate that the current cog is waiting for the cog in which the object that will resolve the future f resides. As opposed to add, addGet does not maintain any irreflexive invariant. Once the value of the future is retrieved, the corresponding chain information is removed from w4 to indicate that the wait is over. We do not have to change the wait-for relation we are carrying forward if we encounter an **await** statement; any of our callers are still blocked and we would have a deadlock if we call back to them.

Finally, we insert the the assertion **assert** cyclefree(w4) prior to every synchronous call or blocking **get** expression. This assertion checks whether or not a directed graph (a possible representation of w4) is a directed acyclic graph (DAG); if not, the assertion will be triggered. We remark that the statement **assert** e in ABS is equivalent to **skip** when e evaluates to true; otherwise they are equivalent to throwing an exception. This is a pitfall for us as exceptions may be caught by already present exception handling and thus interfering with

```
1 w4 = add(w4, Cog(this), Cog(p));
2 assert cyclefree(w4);
3 m = p.getManager(w4);
4 w4 = remove(w4, Cog(this), Cog(p));
```

(a) Assertion at call site

```
1  class Person(CogId id, CogMap cogs, ...)
2  implements PersonI {
3   ...
4   PersonI getManager(Rel w4) {
5     PersonI d = this.getDept(w4);
6     w4 = add(w4, Cog(this), Cog(d));
7     assert cyclefree(w4);
8     PersonI tmp = d.getManager(w4);
9     w4 = remove(w4, Cog(this), Cog(d));
10    return tmp;
11  }
```

(b) Assertion in added method

Fig. 7: After applying the assertion transformation to Fig. 3b

deadlock detection. Our intended semantics on an assert statement that fails is to indicate that a deadlock will occur on further execution of the program. A complete transformation of a program by the rules shown in Fig. 6 is performed by repetition of any rule that matches on the original program.

Note that our treatment of asynchronous calls gives rise to *false positives*: we propagate the current call chain into an asynchronous call, although the previously recorded chain may no longer be current by the time the callee calls back into the chain (if at all). The objects in the call chain that led up to this asynchronous call may long since have become fully available again through termination of the current computation, or partially available due to an **await** statement.

3.2 Example

In this section, we are going to show the assertion transformation applied to a program resulting from a Hide Delegate refactoring and make some observations about when the assertions would detect deadlocks.

Applying the assertion transformation described in Fig. 6 to a refactored program as shown in Fig. 3b results in the code shown in Figs. 7a and 7b. Next, we are going to observe the wait-for relation in the additional method in Fig. 7b invoked through the call site seen in Fig. 7a. Let us first assume the call site is contained in an object c. Consider the sequence of calls $c \xrightarrow{getManager} p \xrightarrow{getDept} d$ where if we are at line 7 in Fig. 7b, the first call has occurred and the last call is about to occur on execution of line 8. We can see that the w4 relation at line 7 in Fig. 7b may contain the two pairs $\langle Cog(c), Cog(p)\rangle$ and $\langle Cog(p), Cog(d)\rangle$. The case where w4 is a singleton set or an empty set refers to the circumstances in which an object calls either itself or another object residing in the same cog because a cog never waits for itself. If we observe the w4 relation we see that $Cog(c) \neq Cog(d)$ must be satisfied; otherwise we have a deadlock. This will also ensure that the assertion in line 7 will not be triggered. An important take-away from this example is not so much the former equation, but that we can record the

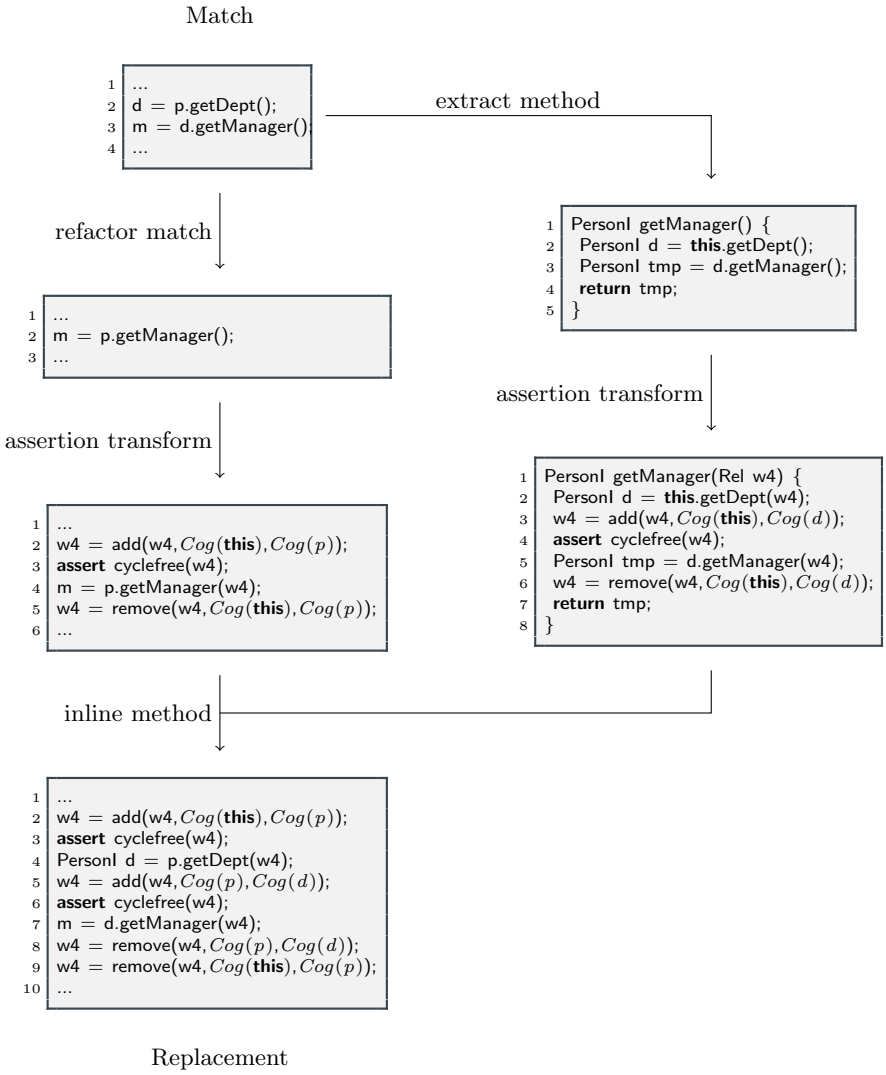


Fig. 8: Clairvoyant Assertion construction

sequence of updates to the w4 relation. For a detailed discussion of all possible object-to-cog allocations for this example see [23].

4 Clairvoyant assertions

Instead of checking if a program may deadlock using the assertion transformation *after* applying a Hide Delegate refactoring, we can produce a *clairvoyant*

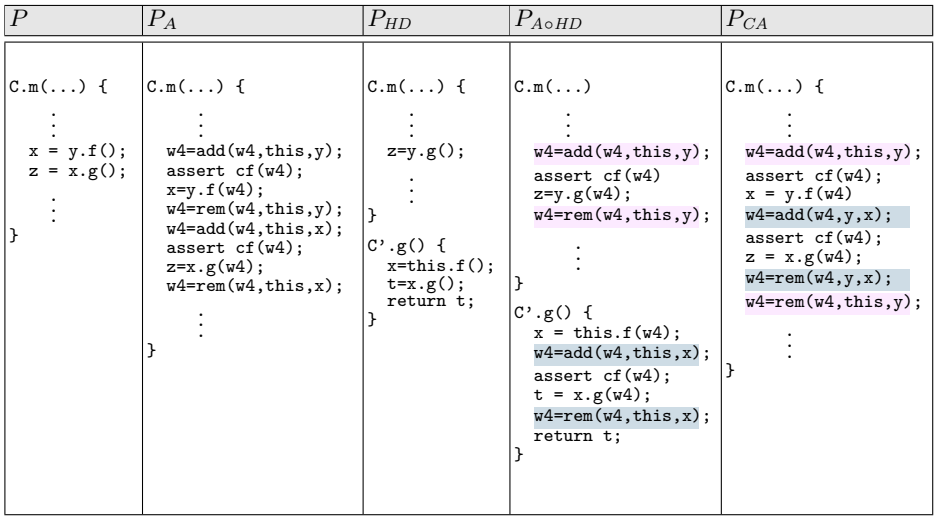


Fig. 9: Effects of the different transformations on P

assertion transformation for Hide Delegate. It constructs assertions and a modification of the wait-for relation such that it predicts occurrences of deadlocks in a refactored program. We define a *clairvoyant assertion* transformation that is almost identical to the assertion transformation from Fig. 6 with one exception: Instead of applying Hide Delegate refactoring, the method calls in Fig. 8 are handled differently. Normally, the call would be transformed into the code shown in Fig. 7a by rule 6f, but it is instead transformed into the replacement code shown at the end of the derivation shown in Fig. 8. This clairvoyant assertion transformation will introduce an assertion that predicts whether the Hide Delegate refactoring when applied to the program will introduce a deadlock.

Fig. 9 shows the effect of the different transformations, where P refers to a program admissible for the Hide Delegate refactoring, P_A the program after the assertion transformation is applied to P, P_{HD} the refactored version of P, $P_{A\circ HD}$ the program after the assertion transformation is applied to P_{HD}, and P_{CA} the program after the clairvoyant assertion transformation is applied to P. Names have been shortened, e.g., `cf` is the `cyclefree` function.

Equivalence between $P_{A\circ HD}$ and P_{CA}. In the following, we informally argue that the effects in $P_{A\circ HD}$ and in P_{CA} wrt the injected assertions are the same, by showing that the wait-for relation is the same at the end of the execution in both programs. Different allocations of objects to cogs will give rise to different executions in a program. In the particular execution shown in Fig. 10, the synchronous calls in P_{CA} (and hence P) are always remote (every synchronous call will be translated into an asynchronous call followed by a blocking **get** operation [16]), i.e., the caller and the callee are always residing in different cogs. This implies that we are in one of two possible scenarios: all three objects are in

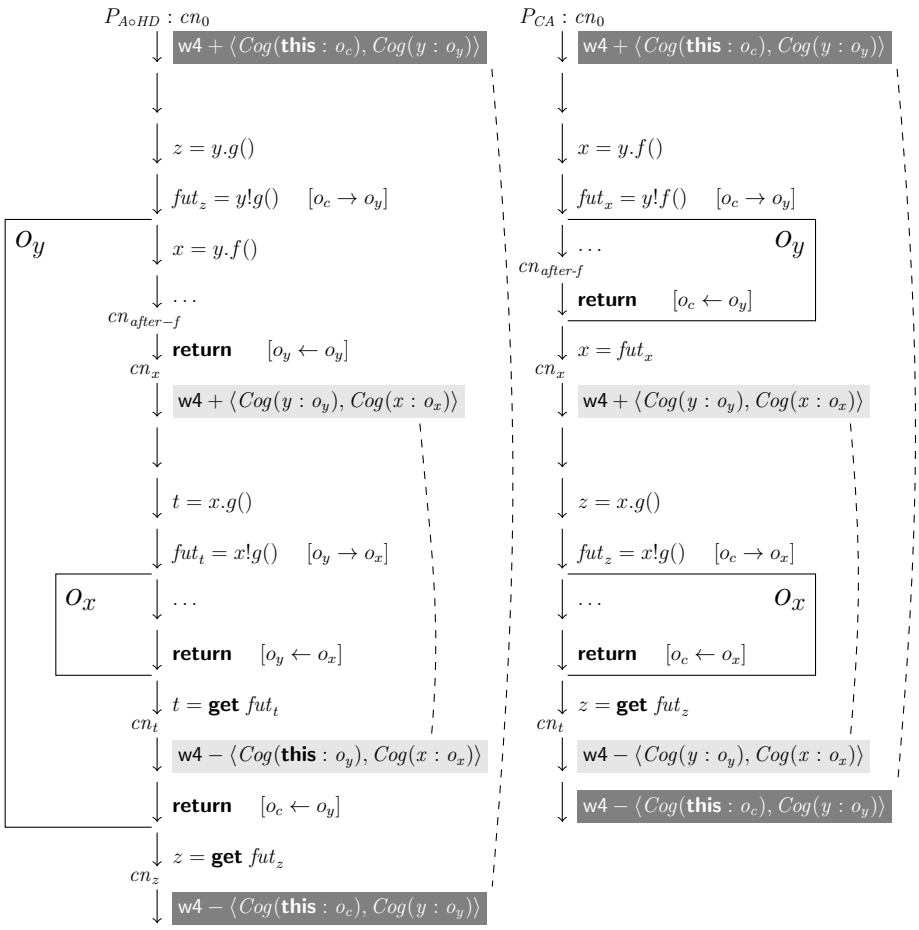


Fig. 10: Equivalence between $P_{A\circ HD}$ and P_{CA} (one execution).

their own cogs, or o_x and o_y are in the same cog. As the execution in P_{CA} uses a remote call from o_y to o_x, it becomes clear that they must be in different cogs, and we find ourselves in the first of the above two possibilities.

We show on the left an execution from the program $P_{A\circ HD}$ and on the right a corresponding execution of the P_{CA}, where the executions start from some state cn_0. Note that for simplicity, the transitions with respect to method binding and object scheduling are not shown in the figure. Due to the strong concurrent semantics of ABS, we also do not have to consider any interleavings. In the figure, we use the operators + and − to manipulate the wait-for relation (w4), where + denotes the addition of a pair of cog identifiers to w4, while − denotes the removal. We also use $var : o$ to indicate in the manipulations the object identity o to which a variable var refers. For method calls, we annotate

the caller and callee objects using $[o_{caller} \rightarrow o_{callee}]$; whereas for returns, we use $[o_{caller} \leftarrow o_{callee}]$ to represent that the method call on the called object terminates and returns back to the caller. Additionally, we indicate the object context the former implies graphically.

This diagram allows us to give the proof idea outlining why the assertions in P_{CA} will always coincide with those in $P_{A \circ HD}$. If we can show that the actual parameters of all w4-manipulations are identical, and that the states of our configurations are equivalent at the end of the refactored code, since the expressions for each pair of assertions are identical, we know that they will give identical results. Although the execution of the refactored program can be different, in a very restricted manner, from that of the original one, they behave equivalently wrt to the w4-relation, which will allow us to draw the necessary conclusions.

As an induction hypothesis, we assume that we have equivalent initial states; and we will see that the same holds for the final states in the end. We note that this is essentially part of the proof that establishes that the equivalence relation $\equiv_{\mathcal{R}}$ between configurations [23] holds between the original program and the refactored program after applying Hide Delegate.

Assuming this, it is obvious that the first assertion checking after the w4-manipulation (or w4-test) uses identical arguments in the respective cn_0-configurations. When both executions reach their respective $cn_{\textit{after-f}}$, it is obvious that either has only exactly executed the method `f()` in object o_y. Hence, when they reach cn_x, the variables `x` in object o_y and the one in object o_c have the same value. From this, we conclude that the next (light gray) w4-test again receives identical objects. Next, either program executes method `g()` on object o_x. Correspondingly, in configurations cn_t, local variables z and t refer to the same value. As `x` and `y` remain unchanged, identical information is removed from w4 in either case (light gray). When control returns in $P_{A \circ HD}$ to o_c, `z` has the same value as `t` before, and hence as `z` in P_{CA}. That means the object states have evolved identically in either execution. The final manipulation of w4 (dark gray) is therefor also identical.

5 KeY-ABS

KeY-ABS [7] is a deductive verification system for the concurrent modelling language ABS [16,14]. It is based on the KeY theorem prover [1,2]. KeY-ABS provides an interactive theorem proving environment and allows one to prove properties of object-oriented and concurrent ABS models. The concurrency model of ABS has been carefully engineered to admit a proof system that is modular and permits to reduce correctness of concurrent programs to reasoning about sequential ones [4,8]. The deductive component of KeY-ABS is an axiomatisation of the operational semantics of ABS in the form of a sequent calculus for first-order dynamic logic for ABS (ABSDL). The rules of the calculus that axiomatise program formulae define a symbolic execution engine for ABS.

Specification and verification of ABS models is done in KeY-ABS dynamic logic (ABSDL). ABSDL is a typed first-order logic plus a box modality: For a

sequence of executable ABS statements s and ABSDL formulae P and Q, the formula $P \rightarrow [s]Q$ expresses: If the execution of s starts in a state where the assertion P holds and the program terminates normally, the assertion Q holds in the final state. Verification of an ABSDL formula proceeds by symbolic execution of s, where state modifications are handled by the update mechanism [2]. An expression of the form $\{u\}$ is called an update application, in which u can be an elementary update of the form $a := t$ which assigns the value of the term t to the program variable a, it can also be a parallel update $u_1 \parallel u_2$ that executes the subupdates u_1 and u_2 in parallel. The semantics of $\{u\}x$ is that an expression x is to be evaluated in the state produced by the update u (the expression x can be a term, a formula, or another update). Given an ABS method m with body mb and a class invariant I, the ABSDL formula $I \rightarrow [mb]I$ expresses that the method m preserves the class invariant. Note that the method body mb may contain **assert** statements. KeY-ABS is able to discharge assertions as vacuous (never fire) or show the open proof at such assertion statements. In ABS, the later one is equivalent to throwing an exception. If the proof can be closed at all **assert** statements, it shows that none of the assertions can be violated.

In this work, we focus on deadlock detection. **assert** statements are added before each of the synchronous calls and are used to predict if the refactored version may cause deadlock while the corresponding synchronous calls are invoked. Since each synchronous call has its own call cycle, it is more suitable to express deadlock cycle in assertion conditions than in class invariants. Consequently, we do not consider the use of class invariants in this setting but assertions. Since the assertion depends on the concrete value of the additional w4 parameter to each method, the required reasoning propagates backwards to call-sites. Eventually this propagation or a proof attempt can result in a contradiction, which indicates a concrete deadlock, although this may be on an infeasible program path. It is then the task of the user to prove this infeasibility, or accept the risk and, for example, resort to testing.

Below is the proof rule for **assert** statement in KeY-ABS.

$$\frac{\Gamma \Longrightarrow \{u\}e = true \qquad \Gamma, \{u\}e = true \Longrightarrow \{u\}[s]\phi, \Delta}{\Gamma \Longrightarrow \{u\}[\textbf{assert } e; s]\phi, \Delta}$$

where Γ and Δ stand for (possibly empty) sets of side formulae. The expression e in the **assert** statement is evaluated according to the update u. The remaining program s can only be verified when the assertion is evaluated to true, i.e., the assertion is not fired. The predicate ϕ is the postcondition of the method and should be proven upon method termination.

6 Related Work

Using theorem provers to discharge assertions is not new. We rely on this existing feature of KeY-ABS, and other comparable tools such as ESC/Java [10,5] deal with them similarly with varying degrees of automation. An alternative to KeY

is Crowbar [18]. Also here we would have to rely on being able to evaluate circularity-queries in the functional fragment of ABS as part of the proof as just like JML the Behavioral Program Logic is not expressive enough to treat them on the level of specifications.

Our encoding in additional data that is only relevant on the specification level is an instance of model variables [6] that model data that goes beyond abstracting the current state. Here, we have introduced data into the program that is intended to be primarily used for reasoning purposes, although they double as concrete program variables for the purpose of runtime checking (a failing assertion indicates an upcoming potential deadlock).

Encoding a static analysis within a theorem proving framework has, to the best of our knowledge, only been done as a proof of concept by Gedell [11]. While his approach also targets the KeY system, the encoding is not in the form of additional data (and properties) thereof in the original program, but as a data structure within the KeY system and an extension of the proof rules for the various syntactic constructs of Java supported by KeY. This has the advantage that no modification of the code is necessary, but requires deeper understanding of the prover framework for the development of the corresponding tactlets. An advantage of our approach is that we are independent from the prover as we embed ourselves within the target language.

That static analysis can in general benefit from relying on theorem provers has already been observed by Manolios and Vroon in [19]. They invoke the ACL2 theorem prover in a controlled manner such that it can be used as a black box when analysing termination. A timeout from the prover gives rise to over-approximation in the static analysis.

The work by Giachino et al. [12] uses an inference algorithm to extract abstract descriptions of methods to detect deadlocks in Core ABS programs, whereas our work captures the potential circular dependencies between cogs by means of a wait-for relation, indicating which method invocations may contribute to deadlock behaviour. Similar to our work, Giachino's approach also over-approximates the occurrence of deadlocks. Albert et al. [3] have developed a comprehensive static analysis based on a may-happen-in-parallel analysis for a very similar language that does not support object groups but treats each object as a singleton member in its own group. They later combined this with a dynamic testing technique that reduces false positives [13].

The work of Kamburjan [17] presents a notion of deadlock for synchronisation on arbitrary boolean conditions in ABS. It supports deadlock detection on condition synchronisation and synchronisation on futures, but it does not consider the cases caused by synchronous method calls as targeted in our work.

Quan et al. formalize refactorings by encoding them as refinement laws in the calculus of refinement of component and object-oriented systems (rCOS) and prove these correct [20], however they do not use a theorem prover and they do not consider concurrent programs.

7 Conclusion

In this paper, we introduce a dynamic deadlock detection mechanism through a program transformation that uses a dependency relation such that assertions can discern deadlocks through inspection of the relation. From the aforementioned transformation we derive a new transformation that manipulates the dependency relation to introduce *clairvoyant* assertions; they predict whether a Hide Delegate refactoring will introduce deadlocks. We argue that in principle our dynamic deadlock detection could be statically discharged by a deductive verification system through resolving the proof goals generated by showing that no assertions trigger.

Discussion and Future Work

While the encoding is certainly useful for runtime checks, using the proof strategies of KeY-ABS to discharge the assertion can be more effective as the proofs cover all the possible execution paths at once.

As a language with interface-based inheritance, it should be clear from the fragments of the transformed ABS programs in Fig. 9 that for every method call there is uncertainty as to which class we are calling into, if the declared type of the callee has more than one implementation. If the classes implementing the same interface have incompatible behaviour, and e.g., only one of the classes will be used at run time, it is again up to the user to provide evidence to the theorem prover that this is the case (and hence eliminate the other classes at this call site). This is however not a particular issue of our approach, but a recurring theme in use of the KeY system, both for ABS and for Java.

We also note that the current version of KeY-ABS does not support **new local**, which is not a problem for the proof, since we explicitly encode the mapping from objects to cogs in the program.

The ABS language does not support object mobility. Integrating this poses a major challenge, since this operation is essentially a side-effect which would mean we would have to give up our model of the deadlock-relation as a purely functional structure. The same will be true for correctly accounting for **await** calls that allow other objects in the same group to make progress concurrently.

To address the shortcoming of false positives (we cannot complete a proof, yet all counterexamples are spurious) in the case where we would need a static analysis to propagate information about the subsequent code backwards into asynchronous calls, we plan to investigate an encoding that uses an oracle in the target language, which ressembles the equivalent encoding of a more precise static analysis in the domain of the prover. It is our goal to remain firmly independant from any particular prover to do our part on encouraging a lively competition between provers.

Clearly working with the code augmented by our assertions has disadvantages for developer in terms of readability. Ideally, such manipulation should be done behind the scenes, preferably in a different view of the model. A natural combination would be to use existing static and dynamic techniques in a first

phase, and discharge any assertions that e.g. are not part of a deadlock-cycle reported by this tools.

The clairvoyant assertions introduced here are specific to the Hide Delegate refactoring. Variations will be necessary to predict negative effects of other refactorings, such as other constellations of deadlocks [23]. We have as yet to implement an automated assertion generation to try out our idea and gauge the currently feasible amount of automation.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (2016), https://doi.org/10.1007/978-3-319-49812-6
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Ulbrich, M. (eds.): Deductive Software Verification: Future Perspectives - Reflections on the Occasion of 20 Years of KeY, Lecture Notes in Computer Science, vol. 12345. Springer (2020), https://doi.org/10.1007/978-3-030-64354-6
3. Albert, E., Flores-Montoya, A., Genaim, S., Martin-Martin, E.: May-happen-in-parallel analysis for actor-based concurrency. ACM Trans. Comput. Log. **17**(2), 11:1–11:39 (2016), https://doi.org/10.1145/2824255
4. Bubel, R., Flores-Montoya, A., Hähnle, R.: Analysis of executable software models. In: Bernardo, M., Damiani, F., Hähnle, R., Johnsen, E.B., Schaefer, I. (eds.) Formal Methods for Executable Software Models - 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014. Lecture Notes in Computer Science, vol. 8483, pp. 1–25. Springer (2014), https://doi.org/10.1007/978-3-319-07317-0_1
5. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005. Lecture Notes in Computer Science, vol. 4111, pp. 342–363. Springer (2005), https://doi.org/10.1007/11804192_16
6. Cheon, Y., Leavens, G.T., Sitaraman, M., Edwards, S.H.: Model variables: cleanly supporting abstraction in design by contract. Softw. Pract. Exp. **35**(6), 583–599 (2005), https://doi.org/10.1002/spe.649
7. Din, C.C., Bubel, R., Hähnle, R.: KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In: Felty, A.P., Middeldorp, A. (eds.) Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction. Lecture Notes in Computer Science, vol. 9195, pp. 517–526. Springer (2015), https://doi.org/10.1007/978-3-319-21401-6_35
8. Din, C.C., Owe, O.: Compositional reasoning about active objects with shared futures. Formal Aspects Comput. **27**(3), 551–572 (2015), https://doi.org/10.1007/s00165-014-0322-y
9. Eilertsen, A.M., Bagge, A.H., Stolz, V.: Safer refactorings. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Part I. Lecture Notes in Computer Science, vol. 9952, pp. 517–531 (2016), https://doi.org/10.1007/978-3-319-47166-2_36

10. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for Java. In: Knoop, J., Hendren, L.J. (eds.) Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 234–245. ACM (2002), https://doi.org/10.1145/512529.512558
11. Gedell, T.: Embedding static analysis into tableaux and sequent based frameworks. In: Beckert, B. (ed.) Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX 2005. Lecture Notes in Computer Science, vol. 3702, pp. 108–122. Springer (2005), https://doi.org/10.1007/11554554_10
12. Giachino, E., Laneve, C., Lienhardt, M.: A framework for deadlock detection in core ABS. Softw. Syst. Model. **15**(4), 1013–1048 (2016), https://doi.org/10.1007/s10270-014-0444-y
13. Gómez-Zamalloa, M., Isabel, M.: Deadlock-guided testing. IEEE Access **9**, 46033–46048 (2021), https://doi.org/10.1109/ACCESS.2021.3065421
14. Hähnle, R.: The abstract behavioral specification language: A tutorial introduction. In: Giachino, E., Hähnle, R., de Boer, F.S., Bonsangue, M.M. (eds.) Formal Methods for Components and Objects - 11th International Symposium, FMCO 2012. Lecture Notes in Computer Science, vol. 7866, pp. 1–37. Springer (2012), https://doi.org/10.1007/978-3-642-40615-7_1
15. Hewitt, C., Bishop, P., Steiger, R.: A Universal Modular ACTOR Formalism for Artificial Intelligence. In: Proc. of the Intl. Joint Conf. on Artificial Intelligence. pp. 235–245. Morgan Kaufmann Publishers Inc. (1973), http://dl.acm.org/citation.cfm?id=1624775.1624804
16. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010. Lecture Notes in Computer Science, vol. 6957, pp. 142–164. Springer (2010), https://doi.org/10.1007/978-3-642-25271-6_8
17. Kamburjan, E.: Detecting deadlocks in formal system models with condition synchronization. Electron. Commun. Eur. Assoc. Softw. Sci. Technol. **76** (2018), https://doi.org/10.14279/tuj.eceasst.76.1070
18. Kamburjan, E., Scaletta, M., Rollshausen, N.: Crowbar: Behavioral symbolic execution for deductive verification of active objects. CoRR **abs/2102.10127** (2021), https://arxiv.org/abs/2102.10127
19. Manolios, P., Vroon, D.: Integrating static analysis and general-purpose theorem proving for termination analysis. In: Osterweil, L.J., Rombach, H.D., Soffa, M.L. (eds.) 28th International Conference on Software Engineering (ICSE 2006). pp. 873–876. ACM (2006), https://doi.org/10.1145/1134285.1134438
20. Quan, L., Qiu, Z., Liu, Z.: Formal use of design patterns and refactoring. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008. Communications in Computer and Information Science, vol. 17, pp. 323–338. Springer (2008), https://doi.org/10.1007/978-3-540-88479-8_23
21. Soares, G., Gheyi, R., Massoni, T.: Automated behavioral testing of refactoring engines. IEEE Trans. Software Eng. **39**(2), 147–162 (2013), https://doi.org/10.1109/TSE.2012.19
22. Steinhöfel, D., Hähnle, R.: Abstract execution. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) Proc. of Formal Methods - The Next 30 Years - Third World Congress. Lecture Notes in Computer Science, vol. 11800, pp. 319–336. Springer (2019), https://doi.org/10.1007/978-3-030-30942-8_20

23. Stolz, V., Pun, V.K.I, Gheyi, R.: Refactoring and active object languages. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Part II. Lecture Notes in Computer Science, vol. 12477, pp. 138–158. Springer (2020), https://doi.org/10.1007/978-3-030-61470-6_9