Refactoring and Active Object Languages*

Volker Stolz[1], Violet Ka I Pun[1], and Rohit Gheyi[2]

[1] Western Norway University of Applied Sciences, Norway
{vsto,vpu}@hvl.no
[2] Federal University of Campina Grande, Brazil
rohit@dsc.ufcg.edu.br

Abstract. Refactorings are important for object-oriented (OO) programs. Actor- and active object programs place an emphasis on concurrency. In this article, we show how well-known OO refactorings such as Hide Delegate, Move Method, and Extract Class interact with a concurrency model that distinguishes between local and remote objects. Refactorings that are straightforward in Java suddenly force the developers to reflect on the underlying assumptions of their actor system. We show that this reflection is primarily necessary for refactorings that add or remove method calls, as well as constructor invocations. We present a general notion of correctness of refactorings in a concurrent setting, and indicate which refactorings are correct under this notion. Finally, we discuss how development tools can assist the developer with refactorings in languages with rich semantics.

1 Introduction

During its life cycle, software may change due to the introduction of new features and enhancements that improve its internal structure, or make its processing more efficient. Systems continue to evolve over time and become more complex as they grow. Developers can take some actions to avoid that, such as code refactoring, a kind of perfective maintenance [1]. The term *Refactoring* was originally coined by Opdyke [2], and popularized in practice by Fowler [3], as the process of changing the internal structure of a program to improve its internal quality while preserving its external behavior.

Over the years refactoring has become a central part of the software development processes, such as eXtreme Programming [4]. Refactorings can be manually applied, which may be time consuming and error prone, or automatically by using implementations of refactoring engines available in IDEs, such as Eclipse, NetBeans, IntelliJ, and JastAdd Refactoring Tools (JRRT) [5]. Refactoring engines may contain a number of refactoring implementations, such as Rename Class, Pull Up Method, and Encapsulate Field. For correctly applying a refactoring, and thus ensuring behavior preservation, the refactoring implementations usually need to consider preconditions, such as checking for naming conflicts. However, defining and implementing refactorings is a nontrivial task

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

since it is difficult to define all preconditions to guarantee that the transformation preserves the program behavior. In fact, proving refactoring correctness for entire languages, such as Java and C, constitutes a challenge [5]. For instance, previous approaches found bugs in refactoring implementations for sequential Java [6, 7].

Fowler advocates that the correctness of refactorings is specified through the unit tests of the software being refactored. This elegantly avoids the discussion of the correctness in all situations of a particular refactoring, which is not given especially in object-oriented programs anyway: even though the required preconditions can be captured statically (see, e.g., [5]), checking them at refactoring-time may yield "don't-know" due to over-approximation and hence limit their applicability [8], or a required notion of equivalence between original and refactored program is impossible to formulate in general due to the different structure of states in both runs [9].

Active object languages [10] for concurrent programs go beyond traditional object oriented method calls. Developers have to actively choose between synchronous and asynchronous method calls. In asynchronous calls, an explicit additional instruction is required to synchronize again with the result. This makes it very obvious to the developers that they are in charge of proper synchronization, and semantic consistency (i.e., to make sure that concurrency within the same object is handled correctly).

The ABS language [11] goes even beyond that distinction: in its component model, objects within the same component, called *concurrent object group* (cog), share the same processor and hence cannot run concurrently. Within the same component, primarily *asynchronous* calls are affected: the caller has to relinquish control to give the component the opportunity to eventually process a pending asynchronous call. In the case of *synchronous* calls *between* components, the calling object does not release control, which hence introduces the potential for deadlocks if the callee directly or indirectly requires a callback into the caller. As the distinction between remote or local is purely semantical, and not visible in the source code, any method call requires careful consideration. This has been addressed, e.g., in [12] through an inference and annotation mechanism, which helps the developer in tracking which objects may be local or remote, and through whole-program static analyses in [13] and [14]. In general however, as other static analyses for object-oriented programs, this inference has to default to "don't know" in case the location of an object cannot statically be determined.

This has direct effects on well-known refactorings in object-oriented programs. Fowler's refactorings [3] often either remove or introduce additional method calls. His refactorings are exclusively on sequential code and in general preserve the behaviour of the application. After our discussion above on the behaviour of actor languages, we can now easily see that these refactorings can have adverse effects when ported to active object languages.

In this article, we are going to investigate those effects in detail for some of Fowler's refactorings. We derive a notion of correctness of a refactoring and show

```
1  class Client {
2    PersonI m,p;
3
4    { ...
5      DeptI d = p.getDept();
6      m = d.getManager();
7      ...
8    }
9
10 }
```

(a) Before

```
11 class Client {
12   PersonI m,p;
13
14   {
15     ...
16     m = p.getManager();
17     ...
18   }
19
20 }
```

```
21 class Person implements PersonI {
22   DeptI d;
23   PersonI getManager() {
24     PersonI p=d.getManager();
25     return p;
26   }
27   DeptI getDept() {
28     return d;
29   }
30 }
```

(b) After

Fig. 1: Before/after Hide Delegate

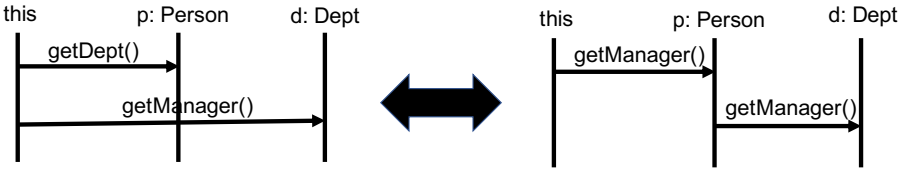


Fig. 2: Sequence diagrams for both scenarios (before/after *Hide Delegate*)

that the refactorings are not correct in general, and identify which refactorings are correct under this notion.

A Motivating Example

Consider Fowler's *Hide Delegate* refactoring and its inverse *Remove Middle Man*. Fig. 1 illustrates a common application of a Java refactoring: assuming that a Client has a reference to a Person, in a good OO design the developer may change person.getDept().getManager() (above broken up into two statements) into a call to a new proxy person.getManager(). While both solutions are considered equivalent in, e.g., Java, the refactoring may introduce a deadlock in the actor setting!

The sequence diagram in Fig. 2 illustrates the difference between the two scenarios: in the *before*-scenario, the client is first communicating with Person, then with Dept. This will never be a problem regardless of how many components are actually involved. In the *after*-scenario however, due to the delegation from the client to Person, the behaviour now depends on the component that provides Dept. If Dept is either in a separate component or shares a component with Person, the programs are equivalent. But in one particular case we have just introduced a new deadlock into the program: if the caller and Dept are in the same component, yet Person is not, then the callback from Person to Dept will deadlock,

			Effect	
:Client	:Person	:Dept	**Before**	**After**
A	A	A	ok	ok
A	A	B	ok	ok
A	B	A	ok	deadlock!
A	B	B	ok	ok
A	B	C	ok	ok

Table 1: Allocation of objects to components A,B,C

since the component hosting the caller and Dept cannot process any request before the synchronous call to Person returns.

Table 1 shows the different possible allocations of objects to components. This information is derived from the ABS operational semantics for (synchronous) method calls. In the remainder of this paper, we investigate this and similar effects in more detail.

The remainder of the paper is structured as follows: Section 2 gives a brief introduction to the ABS language and its component model. Afterwards, we discuss in detail the implications of some prominent Fowler's refactorings on the behaviour of synchronous and asynchronous calls, and outline proofs to show correctness or derive correctness conditions for particular scenarios in Section 3. We then survey Fowler's refactorings as to whether they suggest changes that would result in equivalent Java code, but may change the behaviour in ABS. Finally, in Section 4 we put our work into the context of existing research, and conclude with recommendations for refactoring tool developers.

2 The ABS Language

In this section, we will briefly introduce the ABS language, with active objects and Java-like syntax. We will first discuss the concurrency model of the language, then present the runtime syntax and finally we show the part of the semantics that we used to illustrate the effect of selected refactorings. The complete details of the language can be found in [11].

The concurrency model. ABS is a modeling language for designing, verifying, and executing concurrent software. The language has a Java-like syntax, features with 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 considered as a processor, which can contain 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. As any fields are private, any non-local read or write to fields must be performed explicitly through method invocations. Different cogs can only communicate through asynchronous method calls.

```
class C(MutexI m) {...
  { ...
    await m!enter();
    /* critical section */
    await m!leave();
    ...} }

class Mutex implements MutexI {
  Bool avail = True;
  Unit enter() {await avail; avail = False;}
  Unit leave() {avail = True;} }
```

To provide an intuition, we discuss about the concurrency model with the simple ABS code to the right, which shows the implementation of a class C that acquires

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

Fig. 3: Runtime syntax of ABS [11]; o, f, c are identifiers of object, future, and cog

exclusive access to a critical section by using a block-structured binary lock that is modelled by the class Mutex implementing the straightforward interface MutexI (not shown). The execution of statement **await** m!enter() invokes enter *asynchronously* on object m by putting the method in the process pool of m. The **await** statement suspends the calling method and *releases the control* of the caller object, which can then proceed with the execution of other methods in the process pool. If the statement **await** m!enter() is replaced by a synchronous call m.enter(), the caller object will be *blocked* (does not release control) until the method returns. The enter method on the callee object m will return when the boolean variable avail becomes true. Similar to awaiting an asynchronous method, awaiting a boolean condition will put the currently executing method in the process pool and suspend until the condition becomes true.

Runtime syntax. The runtime syntax is given in Fig. 3. A configuration cn can be empty ϵ or consists of futures, objects, invocation messages and concurrent object groups. The associative and commutative union operator on configurations is denoted by whitespace. A *future* $\mathit{fut}(f, v)$ has an identifier f and a value v (which is $\bot$ when the associated method call has not returned). An object is a term $\mathit{ob}(o, a, p, q)$, where o is the object's identifier, a a substitution representing the object's fields, p an active process, and q a pool of suspended processes. A substitution is a mapping from variable names to values. A process p is idle or consists of a substitution l of local variable bindings and a list s of statements, denoted as $\{l|s\}$. Most of the statements are standard. The statement **suspend** unconditionally releases the processor, suspending the active process. The statement **await** g releases the processor depending on the guard g, which is either Boolean conditions b or return tests $x?$, which evaluates to true if x is a future variable and its value 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 same cog, denoted as **new** $C(e)$, and in a fresh cog, denoted as **new cog** $C(e)$, asynchronous and synchronous method calls, and (pure) expressions e.[3] An invo-

[3] We refer to the semantics in [11], although the ABS surface language has evolved and among other small changes now uses **new local** and **new** instead of **new**/**new cog**.

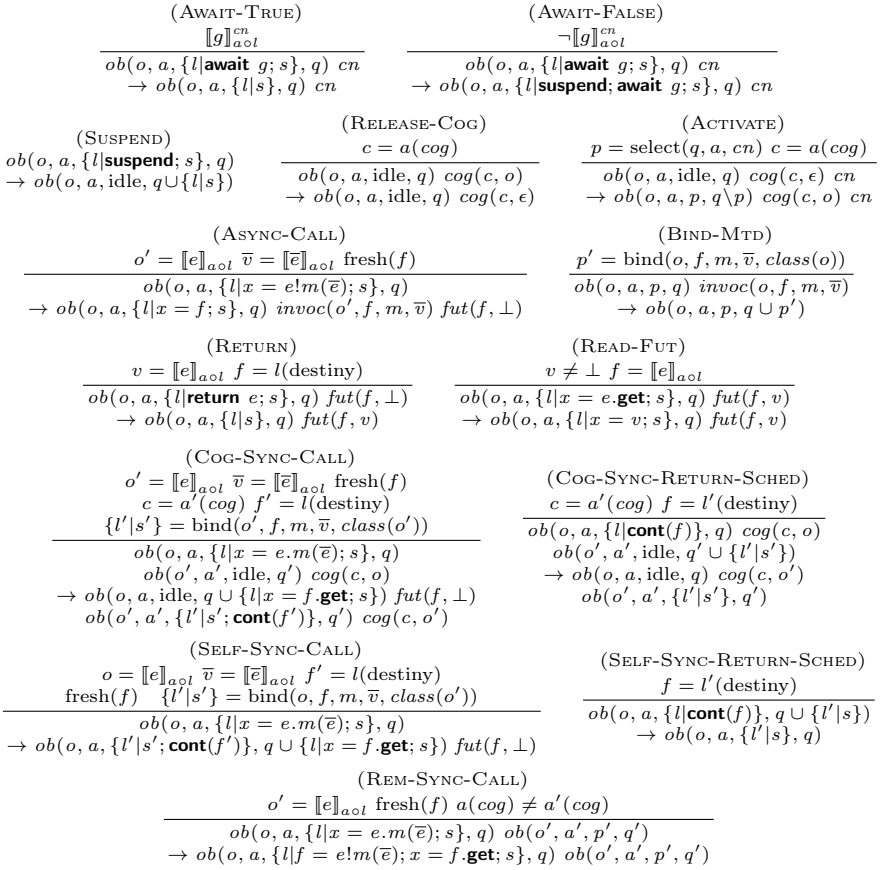


Fig. 4: Part of Semantics of Core ABS [11]

cation 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.

Semantics. Here, we discuss some of the transition rules, given in Fig. 4, of the ABS semantics that we used in evaluation of the refactored ABS programs. Full semantics can be found in [11]. Assignment of an object's fields and a process' local variables is standard and therefore is not shown here. The **await** statements are handled as follows: if the guard g evaluates to *true* in the object's current state, Await-True consumes the statement; otherwise, Await-False appends a **suspend** statement to the process. Rule Suspend puts the active process to the process pool, leaving the processor *idle*, and if a cog's active object is idle, rule Release-Cog releases the cog from the object. When a cog is idle, rule Activate selects a process p from the process pool of an object residing in the cog for

execution. Note that the function *select(q, a, cn)* selects a *ready* process from q; if q is empty or no process is ready, the function returns an idle process [16]. A process is *ready* if it will not directly be resuspended or block the processor.

Rule Async-Call controls the asynchronous communications between objects by sending an invocation message to the callee o' with a new, unique future f (guaranteed by *fresh(f)*), the method name m and actual parameters $\overline{v}$. The value of f is initialised to $\bot$. Rule Bind-Mtd puts the process corresponding to a method invocation in the process pool of the callee. A reserved variable *destiny* local in the method is used to store the identity of the future associated with the call. Rule Return puts the return value of the call into the associated future. Rule Read-Fut retrieves the value from the future f if $v \neq \bot$; otherwise, the reduction on this object is blocked.

The remaining rules in Fig. 4 handle synchronous communication among objects. Rules Cog-Sync-Call and Cog-Sync-Return-Sched are responsible for synchronous calls between two objects residing in the same cog, in which case the possession of the cog is directly transferred between the caller and callee by appending a special **cont** statement at the end of the invoked method. Synchronous self-calls are implemented similarly by rules Self-Sync-Call and Self-Sync-Return-Sched. Rule Rem-Sync-Call handles synchronous calls to an object in a different cog, which is in fact syntactic sugar for an asynchronous call immediately followed by a blocking **get** operation.

3 Refactorings and Their Effects on Concurrency

In this section, we discuss different cases of refactorings that can affect program behaviour in a concurrent setting. First, we define a notion of equivalence of configurations that is suitable for our purpose, as we deal with concurrent systems, and some refactorings may affect the allocation of objects to components (cogs).

Definition 1 (Equivalence of configurations). *Two configurations cn_1 and cn_2 are equivalent, denoted as $cn_1 \equiv_{\mathcal{R}} cn_2$, if and only if for any object o such that $ob(o, a_1, \{l_1|s_1\}, q_1) \in cn_1$ and $ob(o, a_2, \{l_2|s_2\}, q_2) \in cn_2$,*

1. $\forall x \in dom(a_1) \cap dom(a_2) \cdot (x \neq this \wedge x \neq cog) \Rightarrow a_1(x) = a_2(x)$; *and*
2. $\forall x \in dom(l_1) \cap dom(l_2) \cdot l_1(x) = l_2(x)$

Note that this definition specifically mandates that all attributes, local variables and activation state coincide, and the assignment of objects to cogs can be different in the refactored program.

Definition 2 (Notion of refactoring correctness). *Given two equivalent configurations cn_o and cn_r where $cn_o = cn_1\ ob(o, a_1, \{l_1|s_1; s_1'\}, q_1)$, $cn_r = cn_2\ ob(o, a_2, \{l_2|s_2; s_2'\}, q_2)$ and a refactoring $\mathcal{R}f$ such that $s_2 = \mathcal{R}f(s_1)$. We say $\mathcal{R}f$ is* correct *if and only if for all $cn_r \rightarrow^* cn_r'$ where $cn_r' = cn_2'\ ob(o, a_2', \{l_2'|s_2'\}, q_2')$, there exists $cn_o' \rightarrow^* cn_o'$ such that $cn_o' \equiv_{\mathcal{R}} cn_r'$ and $cn_o' = cn_1'\ ob(o, a_1', \{l_1'|s_1'\}, q_1')$.*

We will see that usually we will not achieve unconditional correctness for all refactorings. Most crucially, changes to method calls can result in addition or removal of deadlocks which hence do not result in equivalent configurations. We will capture these side-conditions accordingly.

3.1 *Hide Delegate*

In the following, we revisit the source code before and after applying the *Hide Delegate* refactoring from Fig. 1, in which we elide the obvious, necessary ABS interface declarations PersonI and DeptI. We then discuss the different possible executions (modulo interleaving in the environment) of the original program in Fig. 1(a), and show that while all non-deterministic executions of the refactored program in Fig. 1(b) are contained in the original, there exists a situation that will deadlock after refactoring for a given object-to-component mapping.

Let us first consider the two synchronous method calls in Fig. 1(a). For illustration, we assume object p lives in a cog *different* from the calling object o, while object d lives in the *same* cog. The first call on Line 5 is handled by one of the three rules for synchronous calls, determined by the component-relationship between the calling object and p. If both are in the same component (or even the same object), we first use Cog-Sync-Call (or Self-Sync-Call). In the case where they are in different cogs, we thus proceed with Rem-Sync-Call.

The execution is illustrated in Fig. 5. The rule creates an intermediate *asynchronous* call immediately followed by a **get**, which begins execution through a Async-Call and the Assignment for the intermediate future. Now the current object cannot proceed, and must wait for the environment to Bind-Mtd and Activate the called object, which then immediately Returns, giving the scheduler the opportunity to complete the Read-Fut and Assign to variable d in the caller o. Note that we elided any possible interleavings with cogs in the environment; as the current cog is never released, its state cannot change in between. Next, we continue with the second synchronous method call on Line 6. Since object d resides in the *same cog* as the caller o, we continue with Cog-Sync-Call, which introduces an intermediate future and immediately passes control to d, which after the trivial Return statement in turn Cog-Sync-Return-Scheds and then resumes execution in the caller o with Read-Fut, Assign.

Recall that our correctness criterion is that a *refactored* execution should exist within the original executions. We now study the corresponding scenario after the refactoring, and illustrate in detail how the refactoring introduces a deadlock. Fig. 6 presents the transition steps of the refactored code in Fig. 1(b). Object o executes the single refactored synchronous call on Line 16 to getManager into the cog hosting object p using Rem-Sync-Call, with the corresponding follow-up through Async-Call, Assign, Bind-Mtd, Activate as before. Now, in object p, we see the difference in execution: the proxy getManager now has to make its own Rem-Sync-Call since we assume that d lives in the same cog as o but different from o. However, after the necessary intermediate Async-Call, Assign, Bind-Mtd, it is now *not* possible to execute Activate to continue execution: since object o is blocked on the synchronous call to object p, it does

$$
\begin{array}{rl}
 & cn\ cog(c_1, o)\ cog(c_2, \epsilon)\ ob(o, a, \{l | d = p.gD(); m = d.gM(); s\}, q) \\
 & ob(o_p, a_p, \text{idle}, q_p)\ ob(o_d, a_d, \text{idle}, q_d) \\
\textsc{Rem-Sync-Call} \rightarrow & cn\ cog(c_1, o)\ cog(c_2, \epsilon)\ ob(o, a, \{l | f_1 = p!gD(); d = f_1.\textbf{get}; m = d.gM(); s\}, q) \\
 & ob(o_p, a_p, \text{idle}, q_p)\ ob(o_d, a_d, \text{idle}, q_d) \\
\textsc{Async-Call} \rightarrow & cn\ cog(c_1, o)\ cog(c_2, \epsilon)\ ob(o, a, \{l | f_1 = f_p; d = f_1.\textbf{get}; m = d.gM(); s\}, q) \\
 & ob(o_p, a_p, \text{idle}, q_p)\ ob(o_d, a_d, \text{idle}, q_d)\ invoc(o_p, f_p, gD, \emptyset)\ fut(f_p, \bot) \\
\textsc{Assign} \rightarrow & cn\ cog(c_1, o)\ cog(c_2, \epsilon)\ ob(o, a, \{l[f_1 \mapsto f_p] | d = f_1.\textbf{get}; m = d.gM(); s\}, q) \\
 & ob(o_p, a_p, \text{idle}, q_p)\ ob(o_d, a_d, \text{idle}, q_d)\ invoc(o_p, f_p, gD, \emptyset)\ fut(f_p, \bot) \\
\textsc{Bind-Mtd} \rightarrow & cn\ cog(c_1, o)\ cog(c_2, \epsilon)\ ob(o, a, \{l[f_1 \mapsto f_p] | d = f_1.\textbf{get}; m = d.gM(); s\}, q) \\
 & ob(o_p, a_p, \text{idle}, q_p \cup \{l_p | s_p; \textbf{return}\ e_p\})\ ob(o_d, a_d, \text{idle}, q_d)\ fut(f_p, \bot) \\
 & \qquad \{l_p | s_p; \textbf{return}\ e_p\} = \text{bind}(o_p, f_p, gD, \emptyset, class(o_p)) \\
\textsc{Activate} \rightarrow & cn\ cog(c_1, o)\ cog(c_2, o_p)\ ob(o, a, \{l[f_1 \mapsto f_p] | d = f_1.\textbf{get}; m = d.gM(); s\}, q) \\
 & ob(o_p, a_p, \{l_p | s_p; \textbf{return}\ e_p\}, q_p)\ ob(o_d, a_d, \text{idle}, q_d)\ fut(f_p, \bot) \\
\vdots & \qquad\qquad \vdots \\
\rightarrow & cn\ cog(c_1, o)\ cog(c_2, o_p)\ ob(o, a, \{l[f_1 \mapsto f_p] | d = f_1.\textbf{get}; m = d.gM(); s\}, q) \\
 & ob(o_p, a'_p, \{l'_p | \textbf{return}\ e_p\}, q'_p)\ ob(o_d, a_d, \text{idle}, q_d)\ fut(f_p, \bot) \\
\textsc{Return} \rightarrow & cn\ cog(c_1, o)\ cog(c_2, o_p)\ ob(o, a, \{l[f_1 \mapsto f_p] | d = f_1.\textbf{get}; m = d.gM(); s\}, q) \\
 & ob(o_p, a'_p, \text{idle}, q'_p)\ ob(o_d, a_d, \text{idle}, q_d)\ fut(f_p, v_p) \qquad v_p = [\![e_p]\!]_{a'_p \circ l'_p} \\
\textsc{Read-Fut} \rightarrow & cn\ cog(c_1, o)\ cog(c_2, o_p)\ ob(o, a, \{l[f_1 \mapsto f_p] | d = v_p; m = d.gM(); s\}, q) \\
 & ob(o_p, a'_p, \text{idle}, q'_p)\ ob(o_d, a_d, \text{idle}, q_d)\ fut(f_p, v_p) \\
\textsc{Assign} \rightarrow & cn\ cog(c_1, o)\ cog(c_2, o_p)\ ob(o, a, \{l'' | m = d.gM(); s\}, q)\ ob(o_p, a'_p, \text{idle}, q'_p) \\
 & ob(o_d, a_d, \text{idle}, q_d)\ fut(f_p, v_p) \qquad l'' = l[f_1 \mapsto f_p, d \mapsto v_p] \\
\textsc{Cog-Sync-Call} \rightarrow & cn\ cog(c_1, o_d)\ cog(c_2, o_p)\ ob(o, a, \text{idle}, q \cup \{l'' | m = f_d.\textbf{get}; s\}) \\
 & ob(o_p, a'_p, \text{idle}, q'_p)\ ob(o_d, a_d, \{l_d | s_d; \textbf{return}\ e_d; \textbf{cont}(f)\}, q_d)\ fut(f_p, v_p) \\
 & fut(f_d, \bot) \qquad \{l_d | s_d; \textbf{return}\ e_d\} = \text{bind}(o_d, f_d, gM, \emptyset, class(o_d))\ f = l(\text{destiny}) \\
\vdots & \qquad\qquad \vdots \\
\rightarrow & cn\ cog(c_1, o_d)\ cog(c_2, o_p)\ ob(o, a, \text{idle}, q \cup \{l'' | m = f_d.\textbf{get}; s\}) \\
 & ob(o_p, a'_p, \text{idle}, q'_p)\ ob(o_d, a'_d, \{l'_d | \textbf{return}\ e_d; \textbf{cont}(f)\}, q'_d)\ fut(f_p, v_p)\ fut(f_d, \bot) \\
\textsc{Return} \rightarrow & cn\ cog(c_1, o_d)\ cog(c_2, o_p)\ ob(o, a, \text{idle}, q \cup \{l'' | m = f_d.\textbf{get}; s\}) \qquad v_d = [\![e_d]\!]_{a'_d \circ l'_d} \\
 & ob(o_p, a'_p, \text{idle}, q'_p)\ ob(o_d, a'_d, \{l'_d | \textbf{cont}(f)\}, q'_d)\ fut(f_p, v_p)\ fut(f_d, v_d) \\
\textsc{Cog-Sync-Return-Sched} \rightarrow & cn\ cog(c_1, o)\ cog(c_2, o_p)\ ob(o, a, \{l'' | m = f_d.\textbf{get}; s\}, q)\ ob(o_p, a'_p, \text{idle}, q'_p) \\
 & ob(o_d, a'_d, \text{idle}, q'_d)\ fut(f_p, v_p)\ fut(f_d, v_d) \\
\textsc{Read-Fut} \rightarrow & cn\ cog(c_1, o)\ cog(c_2, o_p)\ ob(o, a, \{l'' | m = v_d; s\}, q)\ ob(o_p, a'_p, \text{idle}, q'_p) \\
 & ob(o_d, a'_d, \text{idle}, q'_d)\ fut(f_p, v_p)\ fut(f_d, v_d) \\
\textsc{Assign} \rightarrow & cn\ cog(c_1, o)\ cog(c_2, o_p)\ ob(o, a, \{l'' [m \mapsto v_d] | s\}, q)\ ob(o_p, a'_p, \text{idle}, q'_p) \\
 & ob(o_d, a'_d, \text{idle}, q'_d)\ fut(f_p, v_p)\ fut(f_d, v_d)
\end{array}
$$

Fig. 5: Execution of the code before *Hide Delegate* (Fig. 1(a)). We abbreviate *getDept* to *gD*, and *getManager* to *gM*. We let o be the object executing Lines 5–6, o_p executing *getDept* and o_d executing *getManager*, and assume $a(cog) = a_d(cog)$, $a(cog) \neq a_p(cog)$.

not release control of the cog it is residing, and consequently object d will never be scheduled. Thus, the three objects in the two cogs are now locked forever in a deadly embrace, as shown in the last configuration in Fig. 6.

Analysing all possible execution scenarios in detail will give us Table 1. We can see that the dynamic behaviour of the source code has to be carefully analysed. Many refactorings are bi-directional; here the application from right to left is Fowler's *Remove Middle Man* refactoring. This example here also illus-

		$cn\ cog(c_1, o)\ cog(c_2, \epsilon)\ ob(o, a, \{l \mid m = p.gM(); s\}, q)\ ob(o_p, a_p, \text{idle}, q_p)\ ob(o_d, a_d, \text{idle}, q_d)$
Rem-Sync-Call	$\rightarrow$	$cn\ cog(c_1, o)\ cog(c_2, \epsilon)\ ob(o, a, \{l \mid f_2 = p!gM(); m = f_2.\textbf{get}; s\}, q)\ ob(o_p, a_p, \text{idle}, q_p)\ ob(o_d, a_d, \text{idle}, q_d)$
Async-Call	$\rightarrow$	$cn\ cog(c_1, o)\ cog(c_2, \epsilon)\ ob(o, a, \{l \mid f_2 = f_p'; d = f_2.\textbf{get}; s\}, q)\ ob(o_p, a_p, \text{idle}, q_p)\ ob(o_d, a_d, \text{idle}, q_d)\ invoc(o_p, f_p', gM, \emptyset)\ fut(f_p', \bot)$
Assign	$\rightarrow$	$cn\ cog(c_1, o)\ cog(c_2, \epsilon)\ ob(o, a, \{l[f_2 \mapsto f_p'] \mid d = f_2.\textbf{get}; s\}, q)\ ob(o_p, a_p, \text{idle}, q_p)\ ob(o_d, a_d, \text{idle}, q_d)\ invoc(o_p, f_p', gM, \emptyset)\ fut(f_p', \bot)$
Bind-Mtd	$\rightarrow$	$cn\ cog(c_1, o)\ cog(c_2, \epsilon)\ ob(o, a, \{l[f_2 \mapsto f_p'] \mid d = f_2.\textbf{get}; s\}, q)\ ob(o_p, a_p, \text{idle}, q_p \cup \{l_p \mid p = d.gM(); \textbf{return}\ e_p'\})\ ob(o_d, a_d, \text{idle}, q_d)\ fut(f_p', \bot)\ \{l_p \mid p = d.gM(); \textbf{return}\ e_p\} = \text{bind}(o_p, f_p', gM, \emptyset, class(o_p))$
Activate	$\rightarrow$	$cn\ cog(c_1, o)\ cog(c_2, o_p)\ ob(o, a, \{l[f_2 \mapsto f_p'] \mid d = f_2.\textbf{get}; s\}, q)\ ob(o_p, a_p, \{l_p \mid p = d.gM(); \textbf{return}\ e_p'\}, q_p)\ ob(o_d, a_d, \text{idle}, q_d)\ fut(f_p', \bot)$
Rem-Sync-Call	$\rightarrow$	$cn\ cog(c_1, o)\ cog(c_2, o_p)\ ob(o, a, \{l[f_2 \mapsto f_p'] \mid d = f_2.\textbf{get}; s\}, q)\ ob(o_p, a_p, \{l_p \mid f_3 = d!gM(); p = f_3.\textbf{get}; \textbf{return}\ e_p'\}, q_p)\ ob(o_d, a_d, \text{idle}, q_d)\ fut(f_p', \bot)$
Async-Call	$\rightarrow$	$cn\ cog(c_1, o)\ cog(c_2, o_p)\ ob(o, a, \{l[f_2 \mapsto f_p'] \mid d = f_2.\textbf{get}; s\}, q)\ ob(o_p, a_p, \{l_p \mid f_3 = f_d'; p = f_3.\textbf{get}; \textbf{return}\ e_p'\}, q_p)\ ob(o_d, a_d, \text{idle}, q_d)\ fut(f_p', \bot)\ invoc(o_d, f_d', gM, \emptyset)\ fut(f_d', \bot)$
Assign	$\rightarrow$	$cn\ cog(c_1, o)\ cog(c_2, o_p)\ ob(o, a, \{l[f_2 \mapsto f_p'] \mid d = f_2.\textbf{get}; s\}, q)\ ob(o_p, a_p, \{l_p[f_3 \mapsto f_d'] \mid p = f_3.\textbf{get}; \textbf{return}\ e_p'\}, q_p)\ ob(o_d, a_d, \text{idle}, q_d)\ fut(f_p', \bot)\ invoc(o_d, f_d', gM, \emptyset)\ fut(f_d', \bot)$
Bind-Mtd	$\rightarrow$	$cn\ cog(c_1, o)\ cog(c_2, o_p)\ ob(o, a, \{l[f_2 \mapsto f_p'] \mid d = f_2.\textbf{get}; s\}, q)\ ob(o_p, a_p, \{l_p[f_3 \mapsto f_d'] \mid p = f_3.\textbf{get}; \textbf{return}\ e_p'\}, q_p)\ ob(o_d, a_d, \text{idle}, q_d \cup gM)\ fut(f_p', \bot)\ fut(f_d', \bot)$ **Deadlocked**

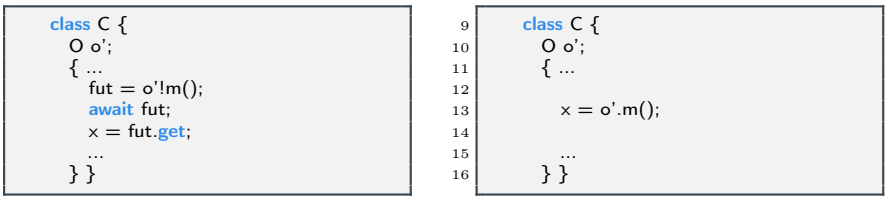
Fig. 6: Execution of the code after *Hide Delegate* (Fig. 1(b)). We abbreviate *getManager* to *gM*. We let o be the object executing Line 16, o_p executing Line 24 and o_d executing *getManager*, and assume $a(cog) = a_d(cog)$, $a(cog) \neq a_p(cog)$.

trates how this refactoring could accidentally *remove* an existing deadlock from a program, and hence cannot immediately fulfil our notion of correctness either.

3.2 *Async-to-Sync* Refactoring

In some situations it may be useful to reduce the amount of concurrency in a program. Fig. 7(a) shows a common idiom in ABS, where we release control while waiting for the asynchronous call to return. This permits **this** object to process other calls in the meantime, though of course this may affect the state of the object. An obvious attempt to reduce such ensuing (mental) complexity would be to use a synchronous call instead. However, whether this is actually safe or not, depends very much on the body of m. Again, a callback into the current component across component boundaries will result in a deadlock. This is compounded by the fact the O o' is only typed by an interface, and additional effort will be required to statically identify the underlying object and then its cog.

We first show the general correctness of this refactoring if both caller and callee are in the *same* cog, and then discuss a similar scenario as in *Hide Delegate*, which leads to a deadlock in the refactored version that does not exist in the



(a) Asynchronous (b) Synchronous

Fig. 7: Asynchronous to synchronous

$$
\begin{array}{rl}
 & cn\ \ cog(c,o)\ \ ob(o,a,\{l|x = o.m(\overline{e});s\},q)\ \ ob(o',a',\mathrm{idle},q') \\
\textsc{Cog-Sync-Call} \rightarrow & cn\ \ cog(c,o')\ \ ob(o,a,\mathrm{idle},q \cup \{l|x = f.\mathbf{get};s\}) \\
 & ob(o',a',\{l'|s';\mathbf{return}\ \ e_m;\mathbf{cont}(f')\},q')\ \ fut(f,\bot) \\
 & \qquad \{l'|s';\mathbf{return}\ \ e_m\} = \mathrm{bind}(o',f',m,\emptyset,class(o'))\ \ f' = l(\mathrm{destiny}) \\
 & \vdots \qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
 & cn\ \ cog(c,o')\ \ ob(o,a,\mathrm{idle},q \cup \{l|x = f.\mathbf{get};s\}) \\
 & ob(o',a'',\{l''|\mathbf{return}\ \ e_m;\mathbf{cont}(f')\},q'')\ \ fut(f,\bot) \\
\textsc{Return} \rightarrow & cn\ \ cog(c,o')\ \ ob(o,a,\mathrm{idle},q \cup \{l|x = f.\mathbf{get};s\})\ \ ob(o',a'',\{l''|\mathbf{cont}(f')\},q'') \\
 & fut(f,v_m) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad v_m = [\![e_m]\!]_{a''ol''} \\
\textsc{Cog-Sync-Return-Sched} \rightarrow & cn\ \ cog(c,o)\ \ ob(o,a,\{l|x = f.\mathbf{get};s\},q)\ \ ob(o',a'',\mathrm{idle},q'')\ \ fut(f,v_m) \\
\textsc{Read-Fut} \rightarrow & cn\ \ cog(c,o)\ \ ob(o,a,\{l|x = v_m;s\},q)\ \ ob(o',a'',\mathrm{idle},q'')\ \ fut(f,v_m) \\
\textsc{Assign} \rightarrow & cn\ \ cog(c,o)\ \ ob(o,a,\{l[x \mapsto v_m]|s\},q)\ \ ob(o',a'',\mathrm{idle},q'')\ \ fut(f,v_m)
\end{array}
$$

Fig. 8: Execution of the synchronous call after refactoring (Fig. 7(b)). We let o be the object executing Line 13, o' executing m, and assume $o \neq o'$, $a(cog) = a'(cog)$

original. However, we will see that the latter is not unconditional as in *Hide Delegate*, but rather (also) depends on the body of m.

We consider the two code fragments in Fig. 7, where Fig. 7(b) is the refactored version of Fig. 7(a), and let o be the calling object and o' be the called object. Let us investigate whether this refactoring is correct wrt. Definition 2, i.e., given equivalent configurations cn_1 and cn_2 before executing Lines 4–6 respectively Line 13, there exists at least one execution in the original program such that the configurations cn_1' and cn_2' after executing Line 6 respectively Line 13 are also equivalent.

For the sake of brevity, as there are many different cases to consider, we look at only one particular case in detail and derive the condition under which the refactoring results in an equivalent program. We essentially distinguish the initial set of different cases by whether we have to invoke Self-Sync-Call (if $o = o'$), Cog-Sync-Call (if $o \neq o'$ but live in the same cog) or Rem-Sync-Call (if o and o' live in different cogs). We only present Cog-Sync-Call in detail, and provide an additional observation on the Rem-Sync-Call case afterwards.

We first consider the refactored program in Fig. 7(b) and assume that $o \neq o'$ but live in the same cog. Fig. 8 shows the detailed execution of this scenario. We start the execution with a Cog-Sync-Call, which introduces an intermediate

$$
\begin{array}{rl}
 & cn\ cog(c,o)\ ob(o,a,\{l|fut = o!m(\overline{e}); \textbf{await}\ fut; x = fut.\textbf{get}; s\}, q)\ ob(o',a',\mathrm{idle},q') \\
\textsc{Async-Call} \rightarrow & cn\ cog(c,o)\ ob(o,a,\{l|fut = f; \textbf{await}\ fut; x = fut.\textbf{get}; s\}, q)\ ob(o',a',\mathrm{idle},q')\ invoc(o',f,m,\emptyset)\ fut(f,\bot) \\
\textsc{Assign} \rightarrow & cn\ cog(c,o)\ ob(o,a,\{l[fut \mapsto f]|\textbf{await}\ fut; x = fut.\textbf{get}; s\}, q)\ ob(o',a',\mathrm{idle},q')\ invoc(o',f,m,\emptyset)\ fut(f,\bot) \\
\textsc{Await-False} \rightarrow & cn\ cog(c,o)\ ob(o,a,\{l[fut \mapsto f]|\textbf{suspend}; \textbf{await}\ fut; x = fut.\textbf{get}; s\}, q)\ ob(o',a',\mathrm{idle},q')\ invoc(o',f,m,\overline{v})\ fut(f,\bot) \\
\textsc{Suspend} \rightarrow & cn\ cog(c,o)\ ob(o,a,\mathrm{idle},q \cup \{l[fut \mapsto f]|\textbf{await}\ fut; x = fut.\textbf{get}; s\})\ ob(o',a',\mathrm{idle},q')\ invoc(o',f,m,\emptyset)\ fut(f,\bot) \\
\textsc{Release-Cog} \rightarrow & cn\ cog(c,\epsilon)\ ob(o,a,\mathrm{idle},q \cup \{l[fut \mapsto f]|\textbf{await}\ fut; x = fut.\textbf{get}; s\})\ ob(o',a',\mathrm{idle},q')\ invoc(o',f,m,\emptyset)\ fut(f,\bot) \\
\textsc{Bind-Mtd} \rightarrow & cn\ cog(c,\epsilon)\ ob(o,a,\mathrm{idle},q \cup \{l[fut \mapsto f]|\textbf{await}\ fut; x = fut.\textbf{get}; s\})\ ob(o',a',\mathrm{idle},q' \cup \{l'|s'; \textbf{return}\ e_m\})\ fut(f,\bot) \\
 & \qquad \{l'|s'; \textbf{return}\ e_m\} = \mathrm{bind}(o',f',m,\emptyset,class(o')) \\
\textsc{Activate} \rightarrow & cn\ cog(c,o')\ ob(o,a,\mathrm{idle},q \cup \{l[fut \mapsto f]|\textbf{await}\ fut; x = fut.\textbf{get}; s\})\ ob(o',a',\{l'|s'; \textbf{return}\ e_m\},q')\ fut(f,\bot) \\
\vdots & \qquad\qquad\qquad\qquad \vdots \\
\rightarrow & cn\ cog(c,o')\ ob(o,a,\mathrm{idle},q \cup \{l[fut \mapsto f]|\textbf{await}\ fut; x = fut.\textbf{get}; s\})\ ob(o',a'',\textbf{return}\ e_m,q'')\ fut(f,\bot) \\
\textsc{Return} \rightarrow & cn\ cog(c,o')\ ob(o,a,\mathrm{idle},q \cup \{l[fut \mapsto f]|\textbf{await}\ fut; x = fut.\textbf{get}; s\})\ ob(o',a'',\mathrm{idle},q'')\ fut(f,v_m) \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad v_m = [\![e_m]\!]_{a''ol''}
\end{array}
$$

$$
\begin{array}{rl}
\textsc{Release-Cog} \rightarrow & cn\ cog(c,\epsilon)\ ob(o,a,\mathrm{idle},q \cup \{l[fut \mapsto f]|\textbf{await}\ fut; x = fut.\textbf{get}; s\})\ ob(o',a'',\mathrm{idle},q'')\ fut(f,v_m) \\
\textsc{Activate} \rightarrow & cn\ cog(c,o)\ ob(o,a,\{l[fut \mapsto f]|\textbf{await}\ fut; x = fut.\textbf{get}; s\}, q)\ ob(o',a'',\mathrm{idle},q'')\ fut(f,v_m) \\
\textsc{Await-True} \rightarrow & cn\ cog(c,o)\ ob(o,a,\{l[fut \mapsto f]|x = fut.\textbf{get}; s\}, q)\ ob(o',a'',\mathrm{idle},q'')\ fut(f,v_m) \\
\textsc{Read-Fut} \rightarrow & cn\ cog(c,o)\ ob(o,a,\{l[fut \mapsto f]|x = v_m; s\}, q)\ ob(o',a'',\mathrm{idle},q'')\ fut(f,v_m) \\
\textsc{Assign} \rightarrow & cn\ cog(c,o)\ ob(o,a,\{l[fut \mapsto f, x \mapsto v_m]|s\}, q)\ ob(o',a'',\mathrm{idle},q'')\ fut(f,v_m)
\end{array}
$$

Fig. 9: Execution of the asynchronous call before refactoring (Fig. 7(a)). We let o be the object executing Lines 4–6, o' executing m, and assume $o \neq o'$, $a(cog) = a'(cog)$

future and yields control to o'. At this point, although there can be interleavings with the environment, wlog. we ignore those, as they cannot interfere with the current cog, except by posting additional tasks into queues. Furthermore, any such interleaving can be simulated in the original program as well. The current cog will proceed evaluating method m through some rule applications $r_0, \ldots, r_m$ and may eventually Return. Note that the refactoring will preserve any potential deadlocks resulting from method m in the original program. We continue the case to completion in the non-deadlocked scenario: the final configuration is easily derived (only) through rules Cog-Sync-Return-Sched, Read-Fut, Assign. Note that in the calling object, the computed value v_m is uniquely determined by the sequence $r_0, \ldots, r_m$ above, as there are no changes in other objects.

We now turn our attention over to the original program in Fig. 7(a) and show in Fig. 9 that we can derive an equivalent state which only differs in the presence

of an explicit, now unused future. Executing Lines 4–6 in the original program can replicate the behaviour of the refactored program in the following way: after the Async-Call and storing the associated future via Assign, execution Suspends until completion of the call. Wlog., we can Release-Cog control and immediately Bind-Mtd and Activate the pending call in o. At this point, we are now entering the execution of m which can proceed exactly with rule sequence $r_0, \ldots, r_m$ as above, which eventually terminates with a Return. The cog hence becomes available through Release-Cog. As the scheduler is not guaranteed to provide any particular behaviour, there exists the behaviour where we Activate the calling object, which now completes with Await-True, Read-Fut, Assign.

Although this is of course not a detailed proof-case, it is easy to see that the resulting configurations are equivalent: the computation of the value of x coincide, and any other state changes can only come from the r-sequence which is identical in both cases. Nonetheless, we would like to motivate the underlying reason for the deadlock in the *Hide Delegate* refactoring, which is only indirectly visible here: assume a program where o and o' are in distinct cogs (which means proceeding with Rem-Sync-Call in the refactored case). Assume further that within $r_0, \ldots, r_m$ there exists a synchronous callback back into the cog of o. In the original program, since o suspends and releases the cog it resides in, by Await-False, this callback can be processed. This execution in the refactored program will however deadlock as the object o blocks on the **get** statement. To summarize, there is again an underlying dynamic condition on the remainder of the code that needs to be checked to ensure correctness.

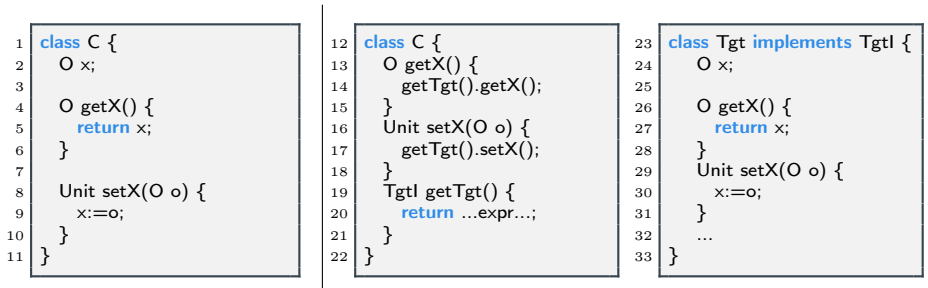
Since the scheduling is non-deterministic, our conversion to a synchronous call *removes* behaviour from the application. There is now only a single scheduling which directly continues into the body of the method.

As for the directionality of this refactoring, while a right-to-left application seemingly enables some degree of concurrency, it is only concurrency on the objects of class C, which as explained initially opens up the caller for possible state changes that may or may not violate assumed invariants by the developer.

3.3 *Inline Method*

The *Inline Method* refactoring is straightforward: within a class, we replace a method call with its body. In the ABS setting, we have two points to consider: as Fowler already points out, this can only be done when the code is not polymorphic. This applies doubly so in ABS, where any variables are only typed by interfaces in the first place. However, it also becomes quickly clear that we only need to consider calls to methods within the same class anyway: a method generally makes use of attributes, and these are private in ABS; so a method from another class cannot easily be inlined but needs to be moved into the current class first (see *Move Method* in Section 3.5 below). Consequently, here we only consider calls for inlining the target **this**.

In the case of a synchronous call, inlining is straightforward and does not affect the behaviour. In fact, from looking at rule Self-Call, it is immediately clear that inlining *is* the semantics of a synchronous self-call.



(a) Before (b) After

Fig. 10: Move Field

3.4 *Move Field*

The *Move Field* refactoring is not as easily applicable in ABS as it is in Java. Declaring the field in a new class is straightforward, however, since all fields are private, as a follow-up we either require the introduction of a getter, relocation of affected methods, or both. We decompose this refactoring into an application of *Self Encapsulate Field*, which first introduces a (synchronous) getter in the current class.

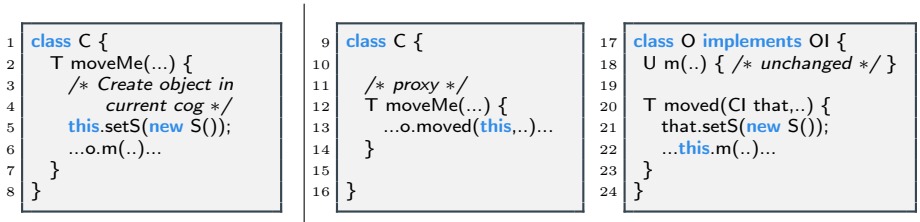
As per the ABS semantics, this introduction results in an equivalent configuration. After that, we can proceed with moving the attribute, introducing a getter, and turning the previous getter into proxy to the getter in the new class. This requires identifying how to reference the target object from the source.

Here, again the ABS language specification makes this easy: as the field was private, the code locations that set this value are immediately identified and we assume for simplicity that we only have to deal with a single setter. After identifying a target, setter and getter now become proxies.

As we have seen before, we now have new (synchronous) calls to objects that may or may not be in the same component as the current object. If the target object is referenced through an attribute in the current class, this is always unproblematic. If the target is derived through a chain of calls, e.g., `getTgt().getX()` (assuming we move attribute `x`), we may have accidentally introduced a deadlock: if `getTgt()` either directly or indirectly calls back (either synchronously or asynchronously) into our object, we will produce a deadlock that did not exist in the original program. Note that if the original program already contained expression `getTgt()`, it already contains the same deadlock, albeit in a different method. The general situation is illustrated in Fig. 10.

3.5 *Move Method*

This refactoring moves a method into a different class, leaving behind a proxy if necessary. Fowler proposes as first step an analysis whether any other features of the class should also be moved. Of the several strategies to handle references to original features, we here focus on passing the source object as a parameter.



(a) Before (b) After

Fig. 11: Move Method

We illustrate a relatively simple case, and will focus our attention on constructor invocations this time, not just method calls.

Fig. 11 shows the initial situation and the refactoring which also leaves behind the proxy. Assuming that the target has been suitably identified as object O o, e.g., if o is a parameter of moveMe, without going through the detailed evaluation by semantic rules, we immediately spot two problematic issues: any reference back to the source object in the refactored code through parameter that has the potential to be a cross-component callback with the associated risk of deadlock, e.g., through the setter, as discussed earlier.

The second issue, which is the novel observation that we can make here is about constructor invocations: in the original in Fig. 11(a), the new object of class S is created in the same cog as the calling object as we use **new**, not **new cog** (the former creates an object in the *same* cog while the latter in a *new* cog). In the refactored code, however, it is now created in a potentially different cog. Note that this is always uncritical when the code uses **new cog**. A similar effect has been observed in Java, where moving a method annotated with @Synchronized can change its synchronization context [17].

As one of the last steps in this refactoring, Fowler suggests to consider removing the remaining proxy and updating call-sites to refer to the new location. This again would have either no effect on deadlocks, or even remove an existing deadlock, as it removes an intermediate call into a potentially different component, but does not otherwise change the sequence of interactions.

3.6 *Extract Class*

The *Extract Class* refactoring is a well-known refactoring simplifying complexity in a class by dividing it into two. Attributes and methods are partitioned between the original program and the new class. It is easy to see that this refactoring primarily relies on *Move Field* and *Move Method*, and hence inherits their properties. Fowler [3] here explicitly suggests that this refactoring in particular *"[improves] the liveness of a concurrent program because it allows you to have separate locks on the two resulting classes"* and points out the potential concurrency issues. In ABS, this issue is made explicit: the split-off class needs to be instantiated through a constructor invocation, at which point the developer has to decide allocating the new object either in a new component, which may

increase concurrency in the future through the introduction of further asynchronous calls, or in the current component. However, as we have now seen, calls across components can lead to deadlocks, if we end up calling back into the current component. Hence the allocation within the same component is always safe, whereas a new component can *only* be used when the split-off class does not have any dependency back into the source class.

3.7 Discussion

We have seen in the above refactorings that the root cause of differences in behaviour in the refactored program are method calls that now cross component boundaries. This may happen either because an invocation is changed (call on a different object, or new call), or because we have moved a constructor invocation into a different class, and hence possibly into a different cog. There is no syntactic criterion to judge changes safe. Even though moving a (local) constructor invocation into a different class through, e.g., the *Move Method* refactoring may be more visible, there is little difference between this and a moved call.

Our *Async-to-Sync* refactoring is an ABS-specific refactoring. From left to right, it may reduce (mental) complexity, at the cost of understanding the safety of the refactored code, first. Applied from right to left, it can be an easy starting point to introduce additional concurrency in the long run. It does not necessarily add concurrency, but enables it by making the code yield, e.g., before a long-running transaction. Also here the effect of yielding needs some up-front analysis and understanding whether any subsequent code after the call may be affected by side-effects while being suspended here. As we have seen in the detailed examples above, there are two main concerns for concurrency: changing method calls can introduce or remove deadlocks, as can moving constructor invocations into a different cog.

In Table 2, we survey a range of refactorings from Fowler [3] for their effect on concurrency. The columns indicate whether a refactoring effects any particular change that we now know to have implications on behaviour. The first six we have studied above, the remainder we classify informally. "Yes/No" indicate whether this change occurs as part of the refactoring, and hence whether careful consideration of effects is required. "Safe" indicates that the change is present in the refactoring, yet will always result in a call to the same object, or in the case of *Replace Method with Method Object* can be kept safe with a local constructor invocation.

While plenty of these refactorings have either no effect or are safe (as calls will still be on the same object), almost any of the major refactorings is affected in some way. Most of the refactoring are innocuous in the Java-world, yet can have surprising effects in ABS, indicating that ABS developers could most likely benefit from dedicated refactoring support.

Suggestions to Tool Developers. It is clear from our discussion that an IDE or tooling for a language like ABS with its rich semantics should assist developers better. Mere syntactical transformations checking structural properties, e.g., on

Refactoring	Change Target of Call?	New Method Call?	Removed Method Call?	New/Moved Constructor?
Inline Method	No	No	Safe	No
Move Method	Yes	No	No	Yes
Move Field	Yes	No	No	Safe
Hide Delegate	Yes	Yes	Yes	Yes
Remove Middle Man	Yes	Yes	Yes	Yes
Extract Class	Yes	Yes	Yes	Yes
Extract Method	No	Safe	No	No
Inline Temp	No	No	No	No
Replace Temp with Query	No	Safe	No	No
Introduce Explaining Variable	No	No	No	No
Split Temporary Variable	No	No	No	No
Replace Method...	No	Safe	No	Safe
Inline Class	Yes	No	Yes	Yes
(Self) Encapsulate Field	No	Safe	No	No

Table 2: Classification of common refactorings whether they affect concurrency

the level of interfaces and classes are of course still essential to guarantee syntactically correct code, but cannot give strong guarantees as to dynamic behaviour. We think that it is important that any further correctness properties also come with a reasonable cost, but do not put undue burden on developers. For example, we feel that any further analysis and checking should be automated, and even though it might be costly in terms of computational power, should avoid requiring any additional input from the developer, e.g., in the form of partial proofs, though light-weight annotations may be acceptable.

We have only focused on program-independent correctness properties here, that we have been able to discharge by showing that mostly identical sequences of evaluation rules can be applied, possibly interleaved with small distinct segments using other semantic rules, that nonetheless do not affect the state that we capture in our notion of correctness. As one would expect for a language with a focus on concurrency, many of the refactorings can introduce potential deadlocks, that fortunately can in principle be tackled through inference of object-to-component allocation. Such inference exists either stand-alone [12], or as part of static deadlock checkers like `DF4ABS` and `DECO` [13,14], and could ideally be re-used to only partially analyse changed code. Currently, the developers' best hope is applying those tools at intermediate stages, though due to their high complexity this may hardly be feasible frequently.

4 Related Work and Conclusion

Related Work. Our work focusses on a dynamic language-feature (object-to-component mapping) that does not exist as such in plain object-oriented languages. The closest related work we are aware of is a precise analysis of the effect of refactorings on concurrent Java code [17], where most notably moving members between classes will change their synchronization context. Agha and Palmskog [18] infer annotations from execution traces that can be used to transform

programs from threads to actors, eliding explicit concurrency primitives. How refactorings affect object lifetime in Rust programs is analysed by Ringdal [19].

Garrido and Meseguer reason about the correctness of refactorings for Java by capturing an executable Java formal semantics in the Maude rewriting logic [20]. As they are concerned with structural refactorings and focus on Pull Up/Push Down and the Rename refactoring, they avoid some of the complexities as to comparing states where refactorings change the bound variables, or produce intermediate states. Schäfer et al. [21] aim for control and data flow preservation, and focus on the *Extract Method* refactoring and decompose it into smaller so-called *micro-refactorings*, for which it is easier to derive or prove properties.

Steinhöfel and Hähnle [22,23] propose Abstract Execution, which generalises Symbolic Execution to partially unspecified programs. They have formalised several of Fowler's refactorings in the KeY framework to prove preservation of a form of behavioural equivalence of Java programs. Careful derivation of preconditions for refactorings is required to prove suitable equivalence of refactored code.

Gheyi et al. [9] use a user-defined equivalence notion between states conforming to different meta-models, e.g., after a refactoring changed the structure of a class. They require an explicit alphabet and a mapping function between added/removed attributes in Alloy models and then check mutual refinement. We conjecture that this could augment our approach, and both alphabet and mapping could be derived from a refactoring and the code that it is applied on.

Eilertsen et al. [8] use assertions to provide runtime warnings in the refactored code in cases where a combination of *Extract* and *Move Method* results in unexpected changes to the object graph. We could easily introduce a similar check on component assignment to provide some protection in those cases where a static safety analysis would have to give up due to imprecision.

Soares et al. [7] propose a technique to automatically identify behavioral changes in a number of refactoring implementations of Eclipse, NetBeans, and JRRT. It uses an automatic program generator (called JDolly [24]) and a tool to automatically detect behavioral changes (called SafeRefactor [25]) to identify a number of bugs in these tools for sequential Java programs. We believe that we may find more behavioral changes when considering concurrent programs, e.g. by following an approach by Pradel et al. [26]. They combine (incomplete) test case generation with (complete) exploration of interleavings through JPF to discover output-diverging substitutes (replacing super-classes with sub-classes). Corresponding necessary generation of test cases for actor systems has been studied e.g. by Li et al. [27].

Rachatasumrit and Kim [28] conduct an empirical study and found that a number of test suites do not test the entities impacted by a refactoring. Test suites do not have a good change coverage. For instance, only 22% of refactored methods and fields are tested by existing regression tests. Mongiovi et al. [29] implement a change impact analyzer tool called Safira, and included it in SafeRefactor. It automatically generates test cases for the entities impacted by the transformation. The tool could find some behavioral changes that

could not be found without Safira. Alves et al. [30] concluded that combining change impact analysis with branch coverage could be highly effective in detecting faults introduced by refactoring edits. A change impact analyzer may be also useful when refactoring concurrent programs. As future work, we intend to evolve Safira to consider transformations applied to concurrent programs.

Conclusion. In this article, we have given an overview of how well-known refactorings from object-oriented programming languages like Java have non-obvious behaviour in actor languages. Here, we have focused on some selected refactorings from Fowler's book [3]. On the example of the ABS active object language, we illustrate the concurrency effects that have to be taken into account.

As the ABS language has a formal semantics [11], we can use it to derive proofs for a suitable notion equivalence of the refactored program. Here, we specify correctness as the refactored behaviour being contained within the original behaviour, in the form of a limited comparison of objects and their attributes/local variables. In the absence of a formal correctness specification of the program being refactored, we find that this gives reasonable expectations as to the effect of the refactoring. Furthermore, from our formal derivations we obtain side conditions that can be checked effectively with existing tools, e.g., related to deadlocks, and can be used to produce counter examples.

Future Work. We have not surveyed refactoring support for other languages such as the Akka library for Scala yet. As a general strategy for identifying high-value targets for closer investigation, it would be useful to first categorize existing static analyses and runtime checks that codify the correctness (usually program-independent properties such as "no crash", "no deadlock"), and then check –as we have done here– to what degree refactorings can affect this. The biggest rewards could be achieved in cases where an expensive analysis or runtime check could be replaced with a modular analysis reasoning only about the performed change in the context of analysis information from the original program.

ABS currently has no refactoring support at all and is in the process of moving towards an Xtext-based compiler infrastructure. This enables deriving a Language Server[4] which should allow us to prototype some of the refactorings with a convenient interface to the outside. This will give us the opportunity to try and integrate some of the inferences as pre-condition checks. A possible feasible approach could be to first transfer the results from the Abstract Execution framework to an active-object language, and then extending it with concurrency.

Another interesting venue of research would be looking into the built-in support for specifying software product lines in ABS through so-called Deltas, which have also already been studied as a subject of refactorings [31]. Deltas specify among other things replacement of methods, but are primarily concerned with evolution, and not refactoring. They could be a convenient vehicle to express and implement refactorings in: an inference and check of the correctness conditions could also be applied to the change specified via a Delta, and hence not

[4] https://microsoft.github.io/language-server-protocol/

only benefit refactorings, but another branch of the ABS language altogether. Developers could then receive warnings if the behaviour of one product diverges from another one, although of course that could be intentional, and is most likely more useful with a proper specification of the program.

References

1. Swanson, E.B.: The dimensions of maintenance. In: Proc. of the Intl. Conf. on Software Engineering. ICSE, IEEE (1976)
2. Opdyke, W.: Refactoring Object-oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign (1992)
3. Fowler, M.: Refactoring - Improving the Design of Existing Code. Addison Wesley object technology series. Addison-Wesley (1999)
4. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley Longman Publishing Company, Inc. (2000)
5. Schäfer, M., de Moor, O.: Specifying and implementing refactorings. In: Object-Oriented Programming, Systems, Languages, and Applications
6. Daniel, B., Dig, D., Garcia, K., Marinov, D.: Automated testing of refactoring engines. In: Proc. of the Foundations of Software Engineering, ACM (2007)
7. Soares, G., Gheyi, R., Massoni, T.: Automated Behavioral Testing of Refactoring Engines. IEEE Transactions on Software Engineering **39**(2) (2013) 147–162
8. Eilertsen, A.M., Bagge, A.H., Stolz, V.: Safer refactorings. In: Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques. Volume 9952 of LNCS., Springer (2016)
9. Gheyi, R., Massoni, T., Borba, P.: An abstract equivalence notion for object models. Electron. Notes Theor. Comput. Sci. **130** (2005) 3–21
10. Boer, F.D., Serbanescu, V., Hähnle, R., Henrio, L., Rochas, J., Din, C.C., Johnsen, E.B., Sirjani, M., Khamespanah, E., Fernandez-Reyes, K., Yang, A.M.: A survey of active object languages. ACM Comput. Surv. **50**(5) (October 2017) 76:1–76:39
11. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Intl. Symp. on Formal Methods for Components and Objects. Volume 6957 of LNCS., Springer (2011)
12. Welsch, Y., Schäfer, J., Poetzsch-Heffter, A.: Location types for safe programming with near and far references. In: Aliasing in Object-Oriented Programming. Types, Analysis and Verification. Volume 7850 of LNCS., Springer (2013)
13. Giachino, E., Laneve, C., Lienhardt, M.: A framework for deadlock detection in core ABS. Software and Systems Modeling **15**(4) (2016) 1013–1048
14. Flores-Montoya, A., Albert, E., Genaim, S.: May-happen-in-parallel based deadlock analysis for concurrent objects. In: Formal Techniques for Distributed Systems. Volume 7892 of LNCS., Springer (2013)
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, Morgan Kaufmann Publishers Inc. (1973)
16. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. In: Software Engineering and Formal Methods, IEEE Computer Society (2004)
17. Schäfer, M., Dolby, J., Sridharan, M., Torlak, E., Tip, F.: Correct refactoring of concurrent Java code. In: Proc. European Conf. on Object-Oriented Programming. Volume 6183 of LNCS., Springer (2010)

18. Agha, G., Palmskog, K.: Transforming threads into actors. In: Principles of Modeling - Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday. Volume 10760 of LNCS., Springer (2018)
19. Ringdal, P.O.: Automated refactorings of Rust programs. Master's thesis, Inst. for Informatics, University of Oslo, Norway (June 2020)
20. Garrido, A., Meseguer, J.: Formal specification and verification of Java refactorings. In: Intl. Workshop on Source Code Analysis and Manipulation, IEEE (2006)
21. Schäfer, M., Verbaere, M., Ekman, T., de Moor, O.: Stepping stones over the refactoring Rubicon. In: Proc. European Conf. on Object-Oriented Programming. Volume 5653 of LNCS., Springer (2009)
22. Steinhöfel, D., Hähnle, R.: Abstract execution. In: Proc. of Formal Methods - The Next 30 Years - Third World Congress. Volume 11800 of LNCS., Springer (2019)
23. Steinhöfel, D.: Abstract Execution: Automatically Proving Infinitely Many Programs. PhD thesis, TU Darmstadt, Dept. of Computer Science (May 2020)
24. Mongiovi, M., Mendes, G., Gheyi, R., Soares, G., Ribeiro, M.: Scaling Testing of Refactoring Engines. In: Software Maintenance and Evolution. ICSME (2014)
25. Soares, G., Gheyi, R., Serey, D., Massoni, T.: Making Program Refactoring Safer. IEEE Software **27**(4) (2010) 52–57
26. Pradel, M., Gross, T.R.: Automatic testing of sequential and concurrent substitutability. In: Intl. Conf. on Software Engineering, ICSE, IEEE (2013)
27. Li, S., Hariri, F., Agha, G.: Targeted test generation for actor systems. In: Proc. European Conf. on Object-Oriented Programming. Volume 109 of LIPIcs., Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018)
28. Rachatasumrit, N., Kim, M.: An empirical investigation into the impact of refactoring on regression testing. In: Intl. Conf. on Software Maintenance. ICSM (2012)
29. Mongiovi, M., Gheyi, R., Soares, G., Teixeira, L., Borba, P.: Making refactoring safer through impact analysis. Science of Computer Programming **93** (2014) 39–64
30. Alves, E.L.G., Massoni, T., de Lima Machado, P.D.: Test coverage of impacted code elements for detecting refactoring faults: An exploratory study. J. Syst. Softw. **123** (2017) 223–238
31. Schulze, S., Richers, O., Schaefer, I.: Refactoring delta-oriented software product lines. In: Aspect-Oriented Software Development, ACM (2013)