Deadlock Analysis of Concurrent Objects – Theory and Practice –

Elena Giachino[1], Carlo A. Grazia[1], Cosimo Laneve[1],
Michael Lienhardt[2], and Peter Y. H. Wong[3]

[1] University of Bologna – INRIA Focus Team, Italy
[2] PPS, Paris Diderot, France
[3] SDL Fredhopper, Amsterdam, The Netherlands

Abstract. We present a framework for statically detecting deadlocks in a concurrent object language with asynchronous invocations and operations for getting the values and releasing the control. Our approach is based on the integration of two static analysis techniques: (i) an inference algorithm to extract abstract descriptions of method's behaviors in the form of behavioral types (contracts), and (ii) an evaluator, which computes a fixpoint semantics, to return finite state models of contracts. A potential deadlock is detected when a circular dependency is found in some state of the model.
We discuss the theory and the prototype implementation of our framework. Our tool is validated on an industrial case study based on the Fredhopper Access Server (FAS) developed by SDL Fredhoppper, an eCommerce software company. In particular we verify one of the core concurrent components of FAS to be deadlock-free.

1 Introduction

Modern systems are designed to support a high degree of parallelism by ensuring that as many system components as possible are operating concurrently. Deadlock represents an insidious and recurring threat when such systems also exhibit a high degree of resource and data sharing. In these systems, deadlocks arise as a consequence of exclusive resource access and circular wait for accessing resources. A standard example is when two processes are exclusively holding a different resource and are requesting access to the resource held by the other. That is, the correct termination of each of the two process activities *depends* on the termination of the other. The presence of a *circular dependency* makes termination not possible.

Deadlocks may be particularly hard to detect in systems where the basic communication operation is asynchronous and the synchronization explicitly occurs when the value is strictly needed. Further difficulties arise in the presence of unbounded (mutual) recursion. A paradigm case is an adaptive system that creates an unbounded number of processes such as server applications. In such systems, process interaction becomes complex and difficult to predict.

`ABS` [2] is an abstract, executable, object-oriented modeling language with a formal semantics, targeting distributed systems. The concurrency model of `ABS` is two-tiered: at the lower level it is similar to that of JCoBox [17] that generalizes the concurrency model of Creol [12] from single concurrent objects to concurrent object groups (COGs). COGs encapsulate synchronous, multi-threaded, shared state computation on a single processor. On top of this level, there is an actor-based model with asynchronous calls, message passing, active waiting, and future types. An essential difference to thread-based concurrency is that task scheduling is *cooperative*, i.e., control switching between tasks of the same

object group happens only at specific scheduling points during the execution, which are explicit in the source code and can be syntactically identified. This allows one to write concurrent programs in a much less error-prone way than in a thread-based model and makes ABS models suitable for static analysis.

We developed a theoretical framework for statically detecting deadlocks in core ABS [11] (a subset of ABS) programs, exploiting and combining results and techniques coming from different well-known theories:

Type theory. We designed an inference system to automatically extract abstract behavioral descriptions pertinent to the deadlock analysis from core ABS code. These descriptions are called *contracts*. This is necessary as analyzing the whole program would be hard and time-consuming, while most part of the code would be irrelevant for deadlock and synchronization behavior, such as the local data and computations. The inference system is constraint-based and uses a standard semiunification technique for solving the set of generated constraints.

Abstract behavioral language. Contracts are defined by a basic behavioral language, that is similar to those ranging from languages for session types to calculi of processes as Milner's CCS or pi-calculus. Likewise, there is also a wide number of theories and tools for verifying their properties. Unlike most techniques on deadlock analysis, however, our behavioral language does not require predefined partial orders and handles dynamic name creation.

Fixpoint Theory. The semantics of contracts is denotational. However, in presence of recursion in the code, a fixpoint may not exist because the underlying model is infinite state (due to creation of new objects). To circumvent this issue, we use a fixpoint technique on models with a limited capacity of name creation. This entails fixpoint existence and finiteness of the models. While we loose in precision, our technique is sound (in some case, it may signal false positives).

We also developed a prototype implementation of our framework. We validated this framework via an industrial case study based on the Fredhopper Access Server (FAS) developed by SDL Fredhoppper[4]. In particular we verified one of the core concurrent components of FAS to be deadlock-free.

The structure of the paper is as follows. In Section 2, we introduce the core ABS language, emphasizing its concurrency model, which is most relevant to this paper. In Section 3, we present the behavioural language for specifying contracts and the inference system for extracting contracts from core ABS programs. In Section 4, we overview the algorithm for computing the contracts into their associated abstract models. In Section 5 we present the implementation of the tool, and its validation against the case study. Related works and concluding remarks are discussed in Section 6.

2 The language core ABS

We begin with a brief presentation of the syntax of core ABS (See Figure 1). Full details of the language, its semantics and its type system, can be found in [11].

In the syntax, an overlined element corresponds to any finite sequence of such element; as usual, an element between square brackets is optional.

A program P is a list of declarations Dl, followed by a main function $\{\overline{T\ x\ ;}\ \ s\}$. Declarations include: *data-types* D and *functions* F constitute the functional part of the

[4] http://sdl.com/products/fredhopper/

P	::= $\overline{Dl}\,\{\,\overline{T\ x\ ;}\ \ s\,\}$	program
Dl	::= $D \mid F \mid I \mid C$	declaration
T	::= $\mathtt{V} \mid \mathtt{D}\langle\overline{T}\rangle \mid \overline{\mathtt{I}}$	type
D	::= $\mathtt{data}\ \mathtt{D}\langle\overline{V}\rangle = \mathtt{Co}\ [\ (\overline{T})\ ]$	data type
F	::= $\mathtt{def}\ T\ \mathtt{f}\ [\langle\overline{T}\rangle](\overline{T\ x}) = e$	function
I	::= $\mathtt{interface}\ \mathtt{I}\ \{\,\overline{S}\,\}$	interface
C	::= $\mathtt{class}\ \mathtt{C}(\overline{T\ x})\ [\mathtt{implements}\ \overline{\mathtt{I}}]\ \{\,\overline{Fl}\ \overline{M}\,\}$	class
Fl	::= $T\ x$	field declatation
S	::= $T\ \mathtt{m}(\overline{T\ x})$	method signature
M	::= $S\{\,\overline{T\ x\ ;}\ \ s\,\}$	method definition
s	::= $\mathtt{skip} \mid s\ ;\ s \mid x = z \mid \mathtt{await}\ g$	statement
	$\mid\ \mathtt{if}\ e\ \{\,s\,\}\ \mathtt{else}\ \{\,s\,\} \mid \mathtt{while}\ e\ \{\,s\,\} \mid \mathtt{return}\ e$	
z	::= $e \mid \mathtt{new}\ [\mathtt{cog}]\ \mathtt{C}\ (\overline{e}) \mid e.\mathtt{m}(\overline{e}) \mid e!\mathtt{m}(\overline{e}) \mid e.\mathtt{get}$	expression with side effects
e	::= $v \mid x \mid \mathtt{this} \mid \mathtt{fun}(\overline{e}) \mid \mathtt{case}\ e\ \{\overline{p \Rightarrow e}\}$	expression
v	::= $\mathtt{null} \mid \mathtt{Co}[(\overline{v})]$	value
p	::= $\_ \mid x \mid \mathtt{null} \mid \mathtt{Co}[(\overline{p})]$	pattern
g	::= $e \mid x? \mid g \wedge g$	guard

Fig. 1. The language core ABS

language, while *interfaces I* and *classes C* constitute its object-oriented part. A type T is a name of either a type variable V used for polymorphism, a datatype with parameters $\mathtt{D}\langle\overline{T}\rangle$ (to type structured data), or an interface I (to type objects). A data type D has a name D and a sequence of parameters $\overline{V}$, and is constructed as a nonempty sequence of type constructors Co with possible parameters $\overline{T}$. Note that data types include primitive types such as Int, Bool, String, as well as complex types such as list of integers $\mathtt{List}\langle\mathtt{Int}\rangle$. The complex type $\mathtt{Fut}\langle T\rangle$ is called *future type* and its values are *futures*. Future type is relevant in core ABS because it is used to type method invocations (that return values of type T). A function F has a return type T, a name f, a sequence of parameters $\overline{T\ x}$, and returns the value of the expression e. An interface I has a name I and a body declaring a sequence of method headers S. A class C has a name C, may implement several interfaces, and declares its fields Fl and its methods M.

A statement s may either be one of the standard operations of a core imperative language or one of the operations for scheduling. Scheduling operations include $\mathtt{await}\ g$, which suspends the execution of the method till the argument, called *guard*, becomes true. Guards include boolean expressions e to be true in order to continue the execution of the method, or future lookups $x?$ that require the value of x to be resolved before resuming the execution of the method, or the conjunction of guards $g \wedge g$.

An expression z may have side effects (may change the state of the system) and is either an object creation $\mathtt{new}\ \mathtt{C}\ (\overline{e})$ in the same group of the creator or an object creation $\mathtt{new}\ \mathtt{cog}\ \mathtt{C}\ (\overline{e})$ in a new group, a method call $e.\mathtt{m}(\overline{e})$ or $e!\mathtt{m}(\overline{e})$, or a get on a expression returning a future value.On the other hand, a *pure* expression e is free of side effects and is either a value v, a variable x, a function application $\mathtt{fun}(\overline{e})$, or a pattern matching $\mathtt{case}\ e\ \{\overline{p \Rightarrow e}\}$. Values include the null object, and structured data $\mathtt{Co}[(\overline{v})]$, while patterns p extend these values with variables x and anonymous variables _.

2.1 The concurrency model of core ABS

We describe informally the concurrency model of core ABS and provide an illustration in the form of a small example. In core ABS, objects belong to a group; a task executing an object method belongs to the object's group. At each point in time there is at most one task per group that is active. The active task may explicitly return the control in order to

```
class Math {
       Int fact_g(Int n){
              if (n==0) { return 1 ; }
              else { Fut<Int> x = this!fact_g(n-1) ;
                     Int m = x.get ;
                     return n*m ; } }
       Int fact_ag(Int n){
              if (n==0) { return 1 ; }
              else { Fut<Int> x = this!fact_ag(n-1) ;
                     await x? ;
                     Int m = x.get ;
                     return n*m ; } }
       Int fact_nc(Int n){
              if (n==0) { return 1 ; }
              else { Math z = new cog Math() ;
                     Fut<Int> x = z!fact_nc(n-1) ;
                     Int m = x.get ;
                     return n*m ; } } }
```

Fig. 2. The class `Math`

let another task of the same group progress. Tasks are created by method invocations: the caller activity continues after the invocation and the called code runs on a different task. Caller and callee synchronize when the result is strictly necessary. In order to decouple method call and returned value, `core ABS` uses futures, i.e., pointers to values that may be not available yet. The access to such values may require waiting for them to be returned.

We illustrate this concurrency model of `core ABS` using three different implementations of the factorial function in an hypothetical class `Math` shown in Figure 2.

The function `fact_g` is the standard definition of factorial: the recursive invocation `this!fact_g(n-1)` is postfixed by a `get` operation that retrieves the value returned by the invocation. Yet, `get` does not allow the task to release the group lock; therefore the task evaluating `this!fact_g(n-1)` is fated to be delayed forever because its object (and, therefore, the corresponding group) is the same of the caller. The function `fact_ag` solves this problem by permitting the caller to release the lock with an explicit `await` operation, before getting the actual value with `x.get`. An alternative solution is defined by the function `fact_nc`, whose code is similar to that of `fact_g`, except for that `fact_nc` invokes `z!fact_nc(n-1)` recursively, where `z` is an object in a new group. This means the task of `z!fact_nc(n-1)` may start without waiting for the release of any lock by the caller.

2.2 Restrictions of `core ABS`

The `core ABS` is a large language and in order to verify the feasibility of our techniques, we considered a subset of its features. Notwithstanding the restrictions we are going to discuss in this section, we were able to verify large commercial codes, such as a core component of FAS discussed in this paper.

Interfaces. In `core ABS` objects are typed with interfaces, which may have several implementations. As a consequence, when a method is invoked, it is in general not possible to statically determine which method will be executed at runtime (dynamic dispatch). This is problematic in our technique because it becomes impossible to associate a unique abstract behavior to method invocations. We avoid this issue by constraining codes to have interfaces implemented by at most one class.

Data types and while loops. In `core ABS`, data types are used to define primitive types (e.g. Booleans) and dynamic structures, such as lists or maps. In particular, dynamic structures

can store an unbounded number of objects and, using a `while` loop, it is possible to invoke methods on these objects according to some ad-hoc protocol. This is problematic as our technique concerns static analysis. Therefore we require the following: *(i)* data types are simply used to store objects of the same class; *(ii)* at each iteration, these objects are manipulated independently (no synchronization with objects in the context is performed), and in an identical manner. A `core ABS` program retaining such properties may be analyzed (as far as deadlocks are concerned) by using *representatives*. Namely, a data type value is abstracted by one of its objects and a `while` loop is abstracted by its body. We remark that, in many usage of dynamic data types and iteration (in particular, in the case study), both conditions hold.

Split synchronizations. `core ABS` allows synchronization primitives (`await` and `get`) to be performed long after the method invocation. Recording the association invocation-synchronization primitives is problematic because it requires the analysis of aliases. To avoid such complexity, we constrain codes to perform the synchronization, when needed, right after the method invocation.

Synchronization on booleans. In addition to synchronization on method invocations, `core ABS` permits synchronizations on Booleans, with the statement `await` g. When g is `False`, the execution of the method is suspended, and when it becomes `True`, the `await` terminates and the execution of the method may proceed. It is possible that the expression g refers to a field of an object that can be modified by another method. In this case, the `await` becomes a synchronization with any possible method that may set the field to `true`. This subtle synchronization pattern is difficult to verify statically. We therefore require `await` statements to be annotated with the dependencies they create. For example, consider the annotated code:

```
class ClientJob(...) {
 Schedules schedules = EmptySet; ConnectionThread thread;
 ...
 Unit executeJob() {
   thread = ...; thread!command(ListSchedule);
   [thread] await schedules != EmptySet;
   ...
 }}
```

The statement `await` compels the task to wait for `schedules` to be set to something different from the empty set. Since `schedules` is a field of the object, any concurrent thread (on that object) may update it. It is not evident how to extract this implicit dependency relation from the guard of `await`. Therefore we constrain the programmer to provide an annotation making explicit the dependency. In the above case, the object that will modify the boolean guard is stored in the variable `thread`. Then the needed annotation is `[thread]`.

Assignments and local variables. Assignments in `core ABS` (as usual in object-oriented languages) may update the fields of objects that are accessed concurrently by other threads, thus leading to rather indeterminate behaviors. Since it is difficult to deal with this feature in a deadlock analysis algorithm, we constrain field assignments to keep the field's record structure unchanged. For instance, if a field contains an object of group a, then that field may be updated only with objects belonging to a (and the correspondence must hold on the fields of the objects recursively). When the field is of a primitive type (`Int`, `Bool`, etc.) this constraint is equivalent to the standard type-correctness. This restriction does not cover local variables of methods, as they can be only accessed by the method where they are declared. In fact it is easy to track local changes in the inference algorithm.

Recursive object structures. In `core ABS`, like in any other object-oriented language, it is possible to define circular object structures, such as an object storing a pointer to itself in

one of its fields. However, our analysis technique associates a finite tree to every object. This allows us to determine the tasks in parallel with high precision. To this end, we forbid object definitions that may produce circular structures.

3 Contracts and the contract inference system

In order to analyze `core ABS` codes, we use abstract descriptions called *contracts*. The syntax of these descriptions uses *record names* X, Y, Z, $\cdots$, and *group names* a, b, $\cdots$. The rules are

$$\begin{array}{lll} \mathbb{r} ::= \_ \mid X \mid a[\overline{\mathtt{f}} : \overline{\mathbb{r}}] \mid a \rightsquigarrow \mathbb{r} & & \text{future record} \\ \mathbb{c} ::= \mathtt{0} \mid (a,a') \mid (a,a')^{\mathtt{w}} \mid \mathtt{C.m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}' \mid \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}' & & \text{contract} \\ \quad \mid \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}' \centerdot (a,a') \mid \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}' \centerdot (a,a')^{\mathtt{w}} \mid \mathbb{c} \fatsemi \mathbb{c} \mid \mathbb{c} + \mathbb{c} & & \end{array}$$

Future records $\mathbb{r}$ may be an empty record $\_$, which corresponds to primitive types, or a record name X, which represents a variable that may be possibly instantiated by substitutions. The future record $a[\overline{\mathtt{f}} : \overline{\mathbb{r}}]$ defines an object with its group name a and the future records of values stored in the fields. The future record $a \rightsquigarrow \mathbb{r}$ specifies that, in order to access to $\mathbb{r}$ one has to acquire the control of the group a (and to release this control once the method has been evaluated). Note that the future record $a \rightsquigarrow \mathbb{r}$ is associated to a method invocation: a is the group of the receiver object of the invoked method.

Contracts $\mathbb{c}$ collect the method invocations and the group dependencies inside statements. A contract may be empty, denoted as $\mathtt{0}$, which expresses that the method behavior is irrelevant for our analysis; or the *dependency pair* (a,a') (resp. $(a,a')^{\mathtt{w}}$) when the dependency is due to a `get` (resp. an `await`) operation on a field or an argument of the method. Alternatively, a contract may also indicate the (type of the) method invocation that may be synchronous $\mathtt{C.m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}'$, or asynchronous $\mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}'$, or asynchronous followed by a `get` $\mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}' \centerdot (a,a')$, or asynchronous followed by an `await` $\mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}' \centerdot (a,a')^{\mathtt{w}}$. The contract $\mathbb{c} \fatsemi \mathbb{c}'$ defines the sequential composition of contracts. The contract $\mathbb{c} + \mathbb{c}'$ defines the abstract behavior of conditionals.

The contract inference algorithm also uses an auxiliary contract (to simplify the rule Seq in Figure 8): $(a,b)^{\mathtt{w}}_{\natural}$. This contract is used to distinguish `await` on Booleans from those on futures. While the two kinds of `await` introduce a dependency pair, the dependency introduced by the `await` on futures may absorb a subsequent dependency produced by a `get`, while this is not possible if the `await` is on Booleans. To separate these two cases, we use $(a,b)^{\mathtt{w}}$ for the `await` on futures and $(a,b)^{\mathtt{w}}_{\natural}$ for the `await` on Booleans. The algorithm introduces the operation $\widehat{\cdot}$ on a contract to remove every "$\natural$" from that contract (This operation is required in the rule Method in Figure 4).

The abstract behavior of a method is defined by a *method contract* $\mathbb{r}(\overline{\mathbb{s}})\ \{\mathbb{c}\}\ \mathbb{r}'$, where $\mathbb{r}$ is the future record of the receiver of the method, $\overline{\mathbb{s}}$ are the future records of the arguments, $\mathbb{c}$ is the contract of the body, and $\mathbb{r}'$ is the future record of the returned object. The subterm $\mathbb{r}(\overline{\mathbb{s}})$ of the method contract is called *header*; $\mathbb{r}'$ is called *returned future record*. Group (and record) names in the header *bind* the group (and record) names in $\mathbb{c}$ and in $\mathbb{r}'$. The header and the returned future record, written $\mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{r}'$, are called *contract signature*.

The contract inference algorithm uses constraints, defined by the following syntax

$$\mathcal{U} ::= \mathtt{true} \mid \mathbb{r} = \mathbb{r}' \mid \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{s} \preceq \mathbb{r}'(\overline{\mathbb{r}'}) \to \mathbb{s}' \mid \mathcal{U} \wedge \mathcal{U} \qquad \text{constraint}$$

where `true` is the constraint that is always true; $\mathbb{r} = \mathbb{r}'$ is a classic unification constraint between terms; $\mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{s} \preceq \mathbb{r}'(\overline{\mathbb{r}'}) \to \mathbb{s}'$ is a *semiunification* constraint; the constraint $\mathcal{U} \wedge \mathcal{U}'$ is the conjunction of $\mathcal{U}$ and $\mathcal{U}'$.

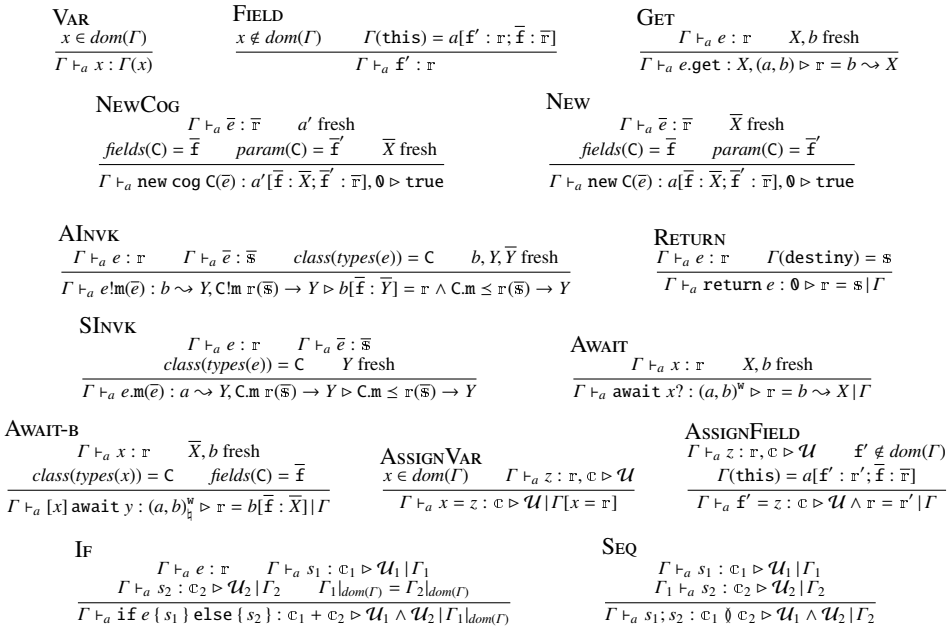

Fig. 3. Contract inference for expressions and statements

Since the contract inference algorithm has a large number of rules, to lighten our presentation, Figures 3 and 4 illustrate a (relevant) subset of them (the complete set of rules is collected in the Appendix). The rules use some auxiliary operators: *fields*(C) and *param*(C) return the sequence of fields and parameters of a class C respectively; *types*(*e*) returns the type of an expression *e*, which is an interface, if *e* is an object; *class*(*I*) returns the unique (see the restriction *Interfaces* in Section 2.2) class implementing *I*; and *mname*($\overline{M}$) returns the sequence of method names in the sequence $\overline{M}$ of method declarations.

Inference statements for pure expressions have the form $\Gamma \vdash_a e : \mathbb{r}$, where Γ is a typing context mapping variables to their records, and methods to their signature; *a* is the group name of the object executing the expression; *e* is the expression; and $\mathbb{r}$ is the inferred record. Constraints and contracts are not generated at this stage.

Inference statements for expressions *z* have the form $\Gamma \vdash_a z : \mathbb{r}, \mathbb{c} \rhd \mathcal{U}$ where Γ, *a*, and $\mathbb{r}$ are as for expressions *e*. The term $\mathbb{c}$ is the contract for *z* created by the inference rules and $\mathcal{U}$ is the generated constraint. Rules NEWCOG and NEW deal with object creation. Rule NEWCOG creates a new group name that is returned in the record of the expression, while NEW uses the same name of the group of `this`. In fact, synchronizations about a `new cog` object have nothing to do with those of `this`. On the contrary, synchronizations on a `new` object will compete with those of `this` and, therefore, as far as deadlocks are concerned, we retains the same group name as that of `this`. It is worth to recall that, in `core ABS`, the creation of an object, either with a `new` or with a `new cog`, amounts to executing the method `init` of the corresponding class, whenever defined (the `new` performs a synchronous invocation, the `new cog` performs an asynchronous one). In turn, the termination of `init` triggers the execution of the method `run`, if present. The method `run` is asynchronously invoked when `init` is absent. Since `init` may be regarded as a method in `core ABS`, in our tool, the inference system explicitly introduces a synchronous invocation to `init` in case of `new` and an asynchronous one in case of `new cog`. However, for

simplicity, we overlook this (simple) issue in the rules New and NewCog, assuming to be always in the case of method `init` (and `run`) absent.

Rules for statements s have the form $\Gamma \vdash_a s : \mathbb{c} \rhd \mathcal{U} \mid \Gamma'$ where Γ, a, s, $\mathbb{c}$ and $\mathcal{U}$ are as before; Γ' is the environment of the method after the execution of the statement. The environment may change because of local variable update. Rule Await deals with the `await` synchronization when applied to simple future lookup $x?$, returning a dependency $(a, b)^{\mathtt{w}}$. In order to correctly associate dependencies to each synchronization, we assume statements of the form `await` $(?x_1 \wedge ?x_2)$ to be decomposed in `await` $?x_1$ `;` `await` $?x_2$. Rule AssignVar is the only rule that changes the environment and manages assignments to local variables of methods. This rule must be compared with AssignField, which deals with assignment of fields. In this case, as we said before, since we do not admit field updates, the rule enforces that the future record of the right-hand-side expression is the same of the field. Rule Return constraints the record of `destiny`, which is an identifier introduced by Method, shown in Figure 4, for storing the return record. Rule Seq defines the sequential composition of contracts. This rule uses an auxiliary operator $\emptyset$, defined in Figure 5, to manage accumulations of dependencies in sequence. With the exception of cases marked (1), (2), (3), and (4) in Figure 5, $\emptyset$ is transformed into a sequential composition $\fatsemi$. Case (1) applies when `await` $x?$ is followed by a x.`get`, where x refers to a field, a global variable or a method argument. Cases (2) and (3) apply when an asynchronous method invocation, whose future reference has been stored into a variable/field x, is immediately followed by a x.`get` and an `await` $x?$ operation, respectively. Case (4) applies when an asynchronous method invocation, whose future reference has been stored into a variable/field x, followed by `await` $x?$ statement, is being composed with a x.`get`. Notice that these rules are defined based on the *split synchronization* restrictions on the usage of `await` and `get` presented in Section 2.2. It is also worth to notice that the contract $\mathbb{c}_1 \emptyset \mathbb{c}_2$ may contain a lot of pairs $(a, b)^{\mathtt{w}}_{\natural}$, which are not modified by $\emptyset$. These pairs are transformed into $(a, b)^{\mathtt{w}}$ by the operation $\widehat{\cdot}$ in the rule Method shown in Figure 4.

(Method)

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{f}} \quad \mathit{param}(\mathtt{C}) = \overline{\mathtt{f}}' \quad a, \overline{X}, \overline{Y}, Z \ \mathit{fresh} \quad \Gamma + \mathtt{this} : a[\overline{\mathtt{ff}}' : \overline{X}] + \overline{\mathtt{x}} : \overline{Y} + \mathtt{destiny} : Z \vdash_a s : \mathbb{c} \rhd \mathcal{U} \mid \Gamma'}{\Gamma \vdash \mathtt{T\ m}\ (\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{s\} : a[\overline{\mathtt{ff}}' : \overline{X}](\overline{Y})\{\widehat{\mathbb{c}}\}\ Z \rhd \mathcal{U} \wedge a[\overline{\mathtt{ff}}' : \overline{X}](\overline{Y}) \to Z = \mathtt{C.m} \quad \text{in } \mathtt{C}}$$

(Class)

$$\frac{\overline{X} \text{ fresh} \quad \Gamma + \overline{\mathtt{ff}}' : \overline{X} \vdash \overline{M} : \overline{\mathbb{C}} \triangleright \overline{\mathcal{U}} \quad \text{in } \mathtt{C}}{\Gamma \vdash \mathtt{class\ C}(\overline{\mathtt{T}}\ \overline{\mathtt{f}})\ \{\overline{\mathtt{T}}'\ \overline{\mathtt{f}}';\ \ \overline{M}\} : \{\mathit{mname}(\overline{M}) \mapsto \overline{\mathbb{C}}\} \rhd \overline{\mathcal{U}}}$$

Fig. 4. Contract rules of method and class declarations

The rules for method and class declarations are defined in Figure 4. In Method, in order to derive the method contract of $\mathtt{T\ m}\ (\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{s\}$, we infer the type of s in an environment extended with `this`, `destiny` (that will be set by `return` statements), and the arguments $\overline{\mathtt{x}}$. The resulting contract $\mathbb{c}$ will be used in the method contract after having transformed every $(a, b)^{\mathtt{w}}_{\natural}$ into $(a, b)^{\mathtt{w}}$ (operation $\widehat{\mathbb{c}}$). The rule Class yields an *abstract class table* that associates a method contract to every method name. It is this abstract class table that is used by our analyzer in Section 4.

For example, according to the inference rules, the methods of `Math` in Figure 2 have the following contracts, once the constraints are solved (we always simplify $\mathbb{c} \fatsemi \mathbb{0}$ into $\mathbb{c}$):

- `fact_g` has contract $a[\,](\_)\ \{\mathbb{0} + \mathtt{Math.fact\_g}\ a[\,](\_) \to \_ \blacksquare (a, a)\}\ \_$. The name a in the header refers to the group name associated to `this` in the code, and binds the occurrences of a in the body. The contract body has a recursive invocation to `fact_g`, which is performed on an object in the same group a and followed by a `get` operation. This operation introduces a dependency pair (a, a). We observe that, if we replace the statement `Fut<Int> x = this!fact_g(n-1)` in `fact_g` with `Math z = new Math() ;`

$$
\begin{aligned}
\mathbb{0} \between \mathbb{c} &= \mathbb{c} \\
\mathbb{c} \fatsemi (a,b) \between \mathbb{c}' &= \mathbb{c} \fatsemi (a,b) \fatsemi \mathbb{c}' \\
\mathbb{c} \fatsemi \mathtt{C.m}\ \mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{r}' \between \mathbb{c}' &= \mathbb{c} \fatsemi \mathtt{C.m}\ \mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{r}' \fatsemi \mathbb{c}' \\
\mathbb{c} \fatsemi (a,b)^{\mathtt{w}} \between \mathbb{c}' &= \begin{cases} \mathbb{c} \fatsemi (a,b)^{\mathtt{w}} \fatsemi \mathbb{c}'' & \text{if } \mathbb{c}' = (a,b) \fatsemi \mathbb{c}'' \\ \mathbb{c} \fatsemi (a,b)^{\mathtt{w}} \fatsemi \mathbb{c}' & \text{otherwise} \end{cases} \qquad (1) \\
\mathbb{c} \fatsemi (a,b)^{\mathtt{w}}_{\natural} \between \mathbb{c}' &= \mathbb{c} \fatsemi (a,b)^{\mathtt{w}}_{\natural} \fatsemi \mathbb{c}' \\
\mathbb{c} \fatsemi \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{r}' \between \mathbb{c}' &= \begin{cases} \mathbb{c} \fatsemi \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{r}' \cdot (a,b) \fatsemi \mathbb{c}'' & \text{if } \mathbb{c}' = (a,b) \fatsemi \mathbb{c}'' \quad (2) \\ \mathbb{c} \fatsemi \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{r}' \cdot (a,b)^{\mathtt{w}} \fatsemi \mathbb{c}'' & \text{if } \mathbb{c}' = (a,b)^{\mathtt{w}} \fatsemi \mathbb{c}'' \quad (3) \\ \mathbb{c} \fatsemi \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{r}' \fatsemi \mathbb{c}' & \text{otherwise} \end{cases} \\
\mathbb{c} \fatsemi \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{r}' \cdot (a,b)^{\mathtt{w}} \between \mathbb{c}' &= \begin{cases} \mathbb{c} \fatsemi \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{r}' \cdot (a,b)^{\mathtt{w}} \fatsemi \mathbb{c}'' & \text{if } \mathbb{c}' = (a,b) \fatsemi \mathbb{c}'' \quad (4) \\ \mathbb{c} \fatsemi \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{r}' \cdot (a,b)^{\mathtt{w}} \fatsemi \mathbb{c}' & \text{otherwise} \end{cases} \\
\mathbb{c} \fatsemi \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{r}' \cdot (a,b) \between \mathbb{c}' &= \mathbb{c} \fatsemi \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{r}' \cdot (a,b) \fatsemi \mathbb{c}' \\
\mathbb{c} + \mathbb{c}' \between \mathbb{c}'' &= \mathbb{c} + \mathbb{c}' \fatsemi \mathbb{c}''
\end{aligned}
$$

Fig. 5. The auxiliary operation $\between$

`Fut<Int> x = z!fact_g(n-1)`, we obtain the same contract as above because the new object is in the same group of `this`.

- `fact_ag` has contract $a[\,](\_)\ \{\mathbb{0} + \mathtt{Math.fact\_ag}\ a[\,](\_) \to \_ \cdot (a,a)^{\mathtt{w}}\}\ \_$. In this case, the presence of an `await` statement in the method body produces a dependency pair $(a,a)^{\mathtt{w}}$. The subsequent `get` operation does not introduce any dependency pair: (a,a) is absorbed by $(a,a)^{\mathtt{w}}$ by definition of $\between$. Intuitively, in this case, the success of `get` is guaranteed, provided the success of the `await` sinchronization.
- `fact_nc` has contract $a[\,](\_)\ \{\mathbb{0} + \mathtt{Math.fact\_nc}\ b[\,](\_) \to \_ \cdot (a,b)\}\ \_$. This method contract differs from the previous ones in that the receiver of the recursive invocation is a free name (i.e., it is not bound by a in the header). This because the recursive invocation is performed on an object of a new group (which is therefore different from a). As a consequence, the dependency pair added by the `get` relates the group a of `this` with the new group b.

Properties. The inference system for contracts possesses the classic soundness and completeness properties. This result is proved in a standard way by (1) defining a type system for contract where method contracts are explicitly provided by programmers; then by (2) demonstrating that this type system is sound with respect to the operational semantics in [11] (subject reduction); and finally by (3) proving that the class table obtained by the inference system yields method contracts that are type correct with respect to (1) (completeness). As regards (1), the type system is very similar to the inference one, but it does not collect constraints. As regards (3), the rules also produce a set of semiunification constraints [10] $\mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{s} \leq \mathbb{r}'(\overline{\mathbb{r}}') \to \mathbb{s}'$ by binding constraints of the form $\mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{s} = \mathtt{C.m}$ (rule Method) with constraints of the form $\mathtt{C.m} \leq \mathbb{r}'(\overline{\mathbb{r}}') \to \mathbb{s}'$ (rules AInvk and SInvk). It is well-known that solving these constraints is undecidable in general [13]. Therefore, it is to be expected that the algorithm loops indefinitely in some cases, which are defined in very ad-hoc ways. In our different tests, we never reached this limitation of our approach.

Theorem 1. *The inference system for contract produces a class table (when the semiunification algorithm terminates) that is sound and complete.*

4 The analysis of contracts

Contracts are inputs to our deadlock analysis technique, which returns finite state models, called *lam* (an acronym for deadLock Analysis Models [7]), where states are relations on group names. For example:

- $[(a,b)]$, $[(a,b),(b,c)]$ is a two-states lam where one state contains the relation $\{(a,b)\}$ and the other state contains $\{(a,b),(b,c)\}$;
- $[(a,b)^{\mathtt{w}}]$ is a one-state lam containing the relation $\{(a,b)^{\mathtt{w}}\}$.

The algorithm takes as input an abstract class table and a main contract, both produced by the inference system. The algorithm applies the standard Knaster-Tarski technique. The critical issue of this technique is that it may create pairs on fresh names at each step, technically speaking, at *every approximant*, because of free names in method contracts that correspond to `new cog`s. As a consequence, the lam model *is not a complete partial order* (the ascending chains of lams may have infinite length and no upper bound). A paradigm example is the model of the *recursive* method contract (of `Math.fact_ng`)

$$\texttt{Math.fact\_nc}\ a[\,](\_)\ \{\mathtt{0} + \texttt{Math.fact\_nc}\ b[\,](\_) \rightarrow \_ \centerdot (a,b)\}\ \_$$

In order to circumvent this issue and to get a decision about deadlock-freedom in a finite number of steps, we use another usual method: running the Knaster-Tarski technique up-to a *fixed approximant*, let us say n, and then using a *saturation argument*. If the n-th approximant is not a fixpoint, then the $(n+1)$-th approximant is computed by *reusing the same group names used by the n-th approximant* (no additional group name is created anymore). Similarly for the $(n+2)$-th approximant till a fixpoint is reached (by straightforward cardinality arguments, the fixpoint does exist, in this case). This fixpoint is called *the saturated state*. For example, in the case of the above contract, the n-approximant returns the single state lam $[(a_1,a_2),\cdots,(a_{n-1},a_n)]$. If we saturate at this stage, the next approximant returns the lam $[(a_1,a_2),\cdots,(a_{n-1},a_n),(a_2,a_2)]$ which contains a circular dependency – the pair (a_2,a_2). This lam is the saturated state. This means that the corresponding program may display a deadlock. In this case, this circularity is a *false positive* that is introduced by the (over)approximation: the original code never manifests a deadlock.

A more detailed account of the algorithm follows; the full definition may be found in [7]. The model of lams is a partial order with a bottom element, which is the single state lam with the emptyset relation. For every syntactic operation on contracts, in particular $+$ and $\fatsemi$, we define a *monotone operation* on the model (an operation is monotone if, whenever it is applied to arguments in the order relation, it returns values in the same order relation). The algorithm analyzing contracts computes an *abstract class table* that associates to every method a function from tuples of group names to *pairs of lams*.

Let us discuss the need for using pairs of lams $\langle \mathcal{W}, \mathcal{W}' \rangle$, where $\mathcal{W}$ and $\mathcal{W}'$ are set of relations on group names. Consider the contract $\mathbb{c} = \texttt{C.m}\ b[\,]()\centerdot(a,b)$. This contract adds the dependency pair (a,b) to the current state. If the method `m` of class `C` only performs a method invocation, let it be `D.n` $b[\,]()$ (without any `get` or `await` synchronization), then the invocation `C.m` $b[\,]()$ does not contribute to the current state with other pairs. However it is possible that `D.n` $b[\,]()$ introduces dependency pairs that affect the *future states* and that have nothing to do with (a,b). The same arguments apply in the cases when `D.n` is a set of states: future dependency pairs are added according to what prescribed by the model of `D.n`. Therefore, in order to augment the precision of our (compositional) abstract semantics, we keep separate the above sets of dependencies in the construction of the abstract model by using pairs of lams. This dichotomy between present and future states is not needed anymore for the main function. In fact, letting $\langle \mathcal{W}_{main}, \mathcal{W}'_{main} \rangle$ be the corresponding model, it is equivalent to the (single) lam $\mathcal{W}_{main} \cup \mathcal{W}'_{main}$. That is, futures are simply additional states to the current ones.

Back to the algorithm analyzing contracts, it builds the abstract class table starting from the first approximant, which associates the function $\lambda\widetilde{a_{\texttt{C,m}}}.\langle \mathtt{0}, \mathtt{0} \rangle$ to every method `C.m`. The next approximant is computed by transforming every entry of the lam class table according to the corresponding contract. When the saturated state is reached, we compute the lam of the main function $\{\ \overline{T\ x\ ;}\ \ s\ \}$. Let $\langle \mathcal{W}_{main}, \mathcal{W}'_{main} \rangle$ be such lam. Then

the program in input is deadlock-free if, for every $W \in \mathcal{W}_{main} \cup \mathcal{W}'_{main}$, $W^{\texttt{get}}$ has no circularity, where $W^{\texttt{get}}$ is defined below.

Definition 1. *Let W be a set of group name dependencies. The* `get`-closure *of W, noted $W^{\texttt{get}}$, is the least set such that*

$$W \in W^{\texttt{get}} \qquad \frac{(a,b) \in W^{\texttt{get}} \quad (b,c) \in W^{\texttt{get}}}{(a,c) \in W^{\texttt{get}}} \qquad \frac{(a,b) \in W^{\texttt{get}} \quad (b,c)^{\texttt{w}} \in W^{\texttt{get}}}{(a,c) \in W^{\texttt{get}}}$$

A set W contains a circularity *if the* `get`*-closure of its dependencies has a pair (a,a).*

As an example, we compute the lams of the methods of class `Math` in Figure 2. The contracts of such methods have been discussed in Section 3. Our analysis algorithm returns

method	first approx.	second approx.	third approx.
`Math.fact_g`	$\lambda a.\langle \mathtt{0}, \mathtt{0} \rangle$	$\lambda a.\langle [(a,a)], \mathtt{0} \rangle$	$\lambda a.\langle [(a,a)], \mathtt{0} \rangle$
`Math.fact_ag`	$\lambda a.\langle \mathtt{0}, \mathtt{0} \rangle$	$\lambda a.\langle [(a,a)^{\texttt{w}}], \mathtt{0} \rangle$	$\lambda a.\langle [(a,a)^{\texttt{w}}], \mathtt{0} \rangle$
`Math.fact_nc`	$\lambda a.\langle \mathtt{0}, \mathtt{0} \rangle$	$\lambda a.\langle [(a,b)], \mathtt{0} \rangle$	$\lambda a.\langle [(a,c),(c,d)], \mathtt{0} \rangle$

We notice that the fixpoints for `Math.fact_g` and `Math.fact_ag` are found at the third iteration. According to the above definition of deadlock-freeness, `Math.fact_g` yields a deadlock, whilst `Math.fact_ag` is deadlock-free. As discussed before, there exists no fixpoint for `Math.fact_nc`. Therefore we may decide to stop at the third iteration and saturate. Saturation of `Math.fact_nc` with $\lambda a.\langle [(a,c),(c,d)], \mathtt{0} \rangle$ yields $\lambda a.\langle [(a,c),(c,c),(c,d)], \mathtt{0} \rangle$ after one iteration, which turns out to be a fixpoint. Since this last lam contains circularity, our algorithm asserts the presence of a deadlock, which is a false positive in this case.

It is worth to notice that saturation might even start at the first approximant (where every method is $\lambda a.\langle \mathtt{0}, \mathtt{0} \rangle$). In this case we get the same answer and the same lam for `Math.fact_g` and `Math.fact_ag`. Whilst for `Math.fact_nc` we get the lam $\lambda a.\langle [(a,b),(b,b)], \mathtt{0} \rangle$, which also displays a circularity. In general, it is possible to augment precision by delaying saturation. Consider in following the abstract class table:

$$\begin{array}{ll} \texttt{C.m}: & a[\,](b[\,],c[\,])\ \{\texttt{C.n}\ b[\,](c[\,]) \to \_ \fatsemi \texttt{C.n}\ c[\,](b[\,]) \to \_\}\ \_ \\ \texttt{C.n}: & a[\,](b[\,])\ \{\texttt{C.p}\ w[\,](a[\,]) \to \_ \fatsemi \texttt{C.p}\ b[\,](w'[\,]) \to \_\}\ \_ \\ \texttt{C.p}: & a[\,](b[\,])\ \{\texttt{C.q}\ b[\,]() \to \_ \cdot (a,b)\}\ \_ \\ \texttt{C.q}: & a[\,]()\ \{\mathtt{0}\}\ \_ \end{array}$$

This class table saturates at the second approximant and uses the same names w and w' in the two invocations of `C.n` inside `C.m`. This will produce a false positive. Saturating at the third approximant, instead, produces a precise response (the program is deadlock-free). We observe that the above abstract class table has a fixpoint at the fourth iteration.

Our technique is correct. We in fact demonstrate the following result.

Theorem 2. *Let $\langle \mathcal{W}_{main}, \mathcal{W}'_{main} \rangle$ be the lams of the main function of a* `core ABS` *program computed with an abstract class table (saturated at the n-th approximant, for some n).*

If no state of $\mathcal{W} \cup \mathcal{W}'$ has a circularity then the program is deadlock-free.

5 The SDA tool and the application to the case study

`ABS` (and, therefore, `core ABS`) comes with a suite [21] that offers a compilation framework, a set of tools to analyze the code, an Eclipse IDE plugin and Emacs mode for

the language. We implemented our static deadlock analysis tool (SDA tool), available at `cs.unibo.it/~laneve/deadlock`, into such suite. The SDA tool is built upon the abstract syntax tree (AST) of the ABS type checker. Therefore we exploit the type informations stored in every node of the tree. This simplifies the implementation of several contract inference rules. The SDA tool is structured in three parts.

1. *Contract and Constraint Generation*. This is performed in three steps: i) the tool first parses the classes of the program and generates a map between interfaces and classes, required for the contract inference of method calls; ii) then it parses again all the classes of the program to generate the initial environment Γ that maps methods to the corresponding method signatures; and iii) it finally parses the AST and, at each node, it applies the contract inference rules in Figures 3, 8, and 4 (and in Appendix).
2. *Constraint Solving* is done by a generic semi-unification solver implemented in Java, following the algorithm defined in [10]. The implementation of that solver is available at `proton.inrialpes.fr/~mlienhar/semi-unification`. When the solver terminates (and no error is found), it produces a substitution that validates the input constraints. Applying this substitution to the generated contracts produces the abstract class table and the contract of the main statement of the program.
3. *Contract Analysis* uses dynamic structures to store states of every method contracts (because states, that record relations, become larger and larger as the analysis progresses). At each stage of the analysis, a number of fresh group names is created and the states are updated according to what prescribed by the contract. A basic operation of the analyzer is the renaming, which is used when computing every approximant. At each step, the tool verifies whether a fixpoint has been reached. Saturation starts when the number of iterations reaches a maximal value (that may be customized by the user). If saturation is reached, since the precision of the algorithm downgrades, the tool signals that the answer (the saturated state) may be imprecise.

5.1 The industrial case study

The Fredhopper Access Server (FAS) is a distributed concurrent object-oriented system that provides search and merchandising services to eCommerce companies. FAS consists of a set of live environments and a single staging environment. A live environment processes queries from client web applications via web services and aims at providing a constant query capacity to client-side web applications. A staging environment is responsible for receiving data updates, and distributing the resulting indices across all live environments according to the *Replication Protocol*. The Replication Protocol has a *SyncServer* module and one *SyncClient* module for each live environment. In turn, the *SyncServer* determines the schedule of replications, as well as its content, while *SyncClient* receives data and configuration updates according to the schedule.

The *SyncServer* communicates to *SyncClient*s by creating *Worker* objects, which serve as the interface to the server-side of the Replication Protocol. On the other hand, *SyncClient*s schedule and create *ClientJob* objects to handle communications to the client-side of the Replication Protocol. When transferring data between the staging and the live environments, it is critical that the data remains *immutable*. To enforce immutability, without interfering the read/write accesses of the staging environment's underlying file system, the *SyncServer* creates a *Snapshot* object that encapsulates a snapshot of the necessary part of the staging environment's file system, and periodically *refreshes* it against the file system. This guarantees immutability of data until it is deemed safe their update. The *SyncServer* uses a *Coordinator* object to determine the safe state in which the *Snapshot* can be refreshed.

Deadlock freedom. As the Replication Protocol is a program with multiple threads interacting concurrently, there are risks of deadlocks. The informal description of the interaction follows.

> For each scheduled replication session, the SyncClient creates a ClientJob. This ClientJob first connects to a Worker (`Acceptor.getConnection`); this connection creates a new `Worker` object. The ClientJob then acquires from the ConnectionThread the next schedule for this replication session (`ClientJob.getSchedules()`) and notifies the SyncClient (`SyncClient.schedule()`). The SyncClient *waits* for the termination of the current ClientJob before proceeding with the next scheduled replication session. In the meantime the current ClientJob receives replication updates from the Worker `ClientJob.sendItems()`.

5.2 The application of SDA to FAS

In order to apply the SDA tool to the case study, we first did few adaptations.

We modified the `core ABS` model such that each interface defined in the model is implemented by at most one class. In particular we have restricted the types of replication items supported by the `core ABS` model to one. This change is adequate for deadlock analysis as these implementations only perform synchronous method calls or function calls with no scheduling point (`await` statements). We have in total removed two implementations of replication item types.

We also removed all circular object structures. For example, in order to keep track of the *number* of `ClientJob` objects active at any given time, the `SyncClient` object keeps a list of references to such objects. On the other hand, each `ClientJob` object keeps a reference to its `SyncClient` object such that it can notify the `SyncClient` at the end of a replication session. We remove `SyncClient`'s reference to `ClientJob` such that `SyncClient` only increments an integer counter when a `ClientJob` is created and decrements the counter when a `ClientJob` object finishes a replication session. We have in total reduced three circular object structures to be non recursive.

Finally, we have annotated every `await` statement on boolean guards with the reference to the object that would resolve the expression to `True`. For example, during the interaction between `ClientJob` and `ConnectionThread`, `ClientJob` asynchronously invokes method `command(ListSchedule)` on `ConnectionThread` to ask the `ConnectionThread` to send all replication schedules, and then waits with the statement `await schedules != EmptySet`, where field `schedules` is subsequently set by `ConnectionThread` to transfer replication schedules on the `ClientJob` object via method `receiveSchedule(Schedules)` (see the code in Section 2.2). In this case we add the annotation `[thread]`, where `thread` is a reference to the `ConnectionThread` object. We have annotated 13 such `await` statements in total.

After these adaptations, we were ready to run the SDA tool. When we iterate till the second approximant and then saturate, we get the answer in Figure 6. That is the tool performs five iterations in order to reach the saturated state. At that stage, there have been produced around 11000 dependencies. The analysis of these dependencies does not reveal any circularity (`Deadlock in Main = false`). This guarantees that the source code is deadlock free.

We conclude with a remark about performance. The constraint inference is pseudo-linear in most of the cases. On the contrary, the fixpoint algorithm is exponential in the number of identifiers in a program. This downgrades a lot the performance. For instance,

```
SATURATION
  Iteration of saturation 1, number of dependencies: 375
  Iteration of saturation 2, number of dependencies: 2462
  Iteration of saturation 3, number of dependencies: 7224
  Iteration of saturation 4, number of dependencies: 10376
  Iteration of saturation 5, number of dependencies: 10865
  . . .
### LOCK INFORMATION RESULTED BY THE ANALYSIS ###
  Saturation: true
  Deadlock in Main: false
```

Fig. 6. The output of the SDA tool for the FAS module.

the analysis of the case study takes in average 2 minutes and 30 seconds on a QuadCore 2.4GHz and Gentoo (Kernel 3.4.9). As we said, the above analysis was run till the second approximant, before saturation. If we run it till the third approximant (which is useless, in this case), the analysis takes more than 24 hours.

6 Related works and conclusions

We have developed a framework for statically detecting deadlocks in `core ABS` and discussed an industrial case study. A preliminary theoretical study was described in [6], while the full theoretical treatment for a sublanguage of `core ABS` can be found in [7]. We observe that our SDA tool is also able to capture *livelocks* (a number of stuck processes that are continuously releasing and acquiring a set of group locks in a circular way) because the underlying theory is similar to the one of deadlocks.

The proposals in the literature that statically analyze deadlocks are largely based on types. Several of them concern process calculi [15, 18–20], but there is some contribution also addressing deadlocks in object-oriented programs [1, 3, 5]. In all these papers, except those of Kobayashi and his collegues, a type system is defined that computes a partial order of the locks in a program and a subject reduction theorem demonstrates that tasks follow this order. On the contrary, our technique does not computes any ordering of locks, thus being more flexible: a computation may acquire two locks in different order at different stages, being correct in our case, but incorrect with the other techniques. The techniques used in [15, 18, 19] are very powerful, since they do not use any predefined partial order of locks and apply to codes with dynamic structures. However the concurrency models are different from that of `ABS` and a precise comparison is matter of future work.

Our work is also related to other static approaches, not based on types. In [4] circular dependencies among processes are detected as erroneous configurations, but dynamic creation of names is not treated.

This study can be extended in several directions. As regards the tool, in the next release, we intend to remove some restrictions, such as the one about synchronizations to be performed immediately after the method invocation. There are standard aliases techniques that we may use to solve this problem. Another extension, this one is more relevant, is to implement a contract analysis technique which is different from that of the current version (fixpoint technique). We have already developed the theory [8, 9], which extends the theory of permutation to lams, and we are going to implement it.

Our technique seems also adequate for deadlocks due to process synchronizations, as those in process calculi [15, 16], and we intend to study in detail this application. In this

case, names are *channel names* and there is a prototype analyzer for deadlocks using a different technique [14]. We will compare it with our SDA tool.

References

1. M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.*, 28, 2006.
2. *The ABS Language Specification*, ABS version 1.2.0 edition, Sept. 2012. `http://tools.hats-project.eu/download/absrefmanual.pdf`.
3. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe program.: preventing data races and deadlocks. In *Proc. OOPSLA '02*, pages 211–230. ACM, 2002.
4. R. Carlsson and H. Millroth. On cyclic process dependencies and the verification of absence of deadlocks in reactive systems, 1997.
5. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *In PLDI 03: Programming Language Design and Implementation*, pages 338–349. ACM, 2003.
6. E. Giachino and C. Laneve. Analysis of deadlocks in object groups. In *FMOODS/FORTE*, volume 6722 of *Lecture Notes in Computer Science*, pages 168–182. Springer-Verlag, 2011.
7. E. Giachino and C. Laneve. Deadlock and livelock analysis in concurrent objects with futures. Technical Report. Available at `www.cs.unibo.it/~laneve`, 2012.
8. E. Giachino and C. Laneve. A beginner's guide to the deadLock Analysis Model. In *TGC*, Lecture Notes in Computer Science. Springer-Verlag, 2013.
9. E. Giachino and C. Laneve. Mutations, flashbacks and deadlocks. Submitted. Available at `www.cs.unibo.it/~laneve`, 2013.
10. F. Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, Apr. 1993.
11. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 142–164. Springer-Verlag, 2011.
12. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and System Modeling*, 6(1):35–58, Mar. 2007.
13. A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. *Inf. Comput.*, 102(1):83–101, 1993.
14. N. Kobayashi. Type systems for concurrent programs. In *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *Lecture Notes in Computer Science*, pages 439–453. Springer, 2003.
15. N. Kobayashi. A new type system for deadlock-free processes. In *Proceedings of the 17th international conference on Concurrency Theory*, CONCUR'06, pages 233–247, Berlin, Heidelberg, 2006. Springer-Verlag.
16. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, ii. *Inf. and Comput.*, 100:41–77, 1992.
17. J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *European Conference on Object-Oriented Programming (ECOOP'10)*, volume 6183 of *Lecture Notes in Computer Science*, pages 275–299. Springer-Verlag, June 2010.
18. K. Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In *Programming Languages and Systems*, volume 5356 of *LNCS*, pages 155–170. Springer, 2008.
19. K. Suenaga and N. Kobayashi. Type-based analysis of deadlock for a concurrent calculus with interrupts. In *Programming Languages and Systems*, volume 4421 of *LNCS*, pages 490–504. Springer, 2007.
20. V. T. Vasconcelos, F. Martins, and T. Cogumbreiro. Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In *Proc. PLACES'09*, volume 17 of *EPTCS*, pages 95–109, 2009.
21. P. Y. H. Wong, E. Albert, R. Muschevici, J. Proença, J. Schäfer, and R. Schlatte. The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *Journal on Software Tools for Technology Transfer*, 14(5):567–588, 2012.

A APPENDIX

A.1 Contract Inference Rules

The inference algorithm is initialized by first generating the mapping between interfaces and classes, and generating an initial environment Γ mapping all methods to a method signature.

Contract inference for programs, classes and methods is presented in Figure 7. Contract inference statements for programs and classes have the form $\Gamma \vdash E : \mathbb{R} \triangleright \mathcal{U}$ where Γ is the typing environment; E is the inferred element; $\mathbb{R}$ is a mapping from method names to their contracts; and $\mathcal{U}$ is the set of generated constraints. Contract inference statements for methods have the form $\Gamma \vdash M : \mathbb{c} \triangleright \mathcal{U}$, which is similar to the one for classes, except that methods are directly associated with their contract, not with a mapping.

$$\textsc{Program}\quad \frac{\Gamma \vdash \overline{C} : \overline{\mathbb{R}} \triangleright \overline{\mathcal{U}} \qquad \overline{X}\ \text{fresh} \qquad \Gamma; \overline{x} : \overline{X} \vdash s : \mathbb{c} \triangleright \mathcal{U} \mid \Gamma'}{\Gamma \vdash \overline{D}\ \overline{F}\ \overline{I}\ \overline{I}\{\overline{T}\ \overline{x}\ ;\ s\} : \overline{\mathbb{R}} \triangleright \overline{\mathcal{U}}}$$

$$\textsc{Class}\quad \frac{\overline{X}\ \text{fresh} \qquad \Gamma + \overline{\texttt{ff}}' : \overline{X} \vdash \overline{M} : \overline{\mathbb{C}} \triangleright \overline{\mathcal{U}} \qquad \text{IN C}}{\Gamma \vdash \texttt{class C}(\overline{\texttt{T}}\ \overline{\texttt{f}})\ \{\overline{\texttt{T}}'\ \overline{\texttt{f}}'\ ;\ \overline{M}\} : \{\mathit{mname}(\overline{M}) \mapsto \overline{\mathbb{C}}\} \triangleright \overline{\mathcal{U}}}$$

$$\textsc{Method}\quad \frac{\mathit{fields}(\texttt{C}) = \overline{\texttt{f}} \qquad \mathit{param}(\texttt{C}) = \overline{\texttt{f}}' \qquad a, \overline{X}, \overline{Y}, Z\ \mathit{fresh} \qquad \Gamma + \texttt{this} : a[\overline{\texttt{ff}}' : \overline{X}] + \overline{\texttt{x}} : \overline{Y} + \texttt{destiny} : Z \vdash_a s : \mathbb{c} \triangleright \mathcal{U} \mid \Gamma'}{\Gamma \vdash \texttt{T m}\ (\overline{\texttt{T}}\ \overline{\texttt{x}})\{s\} : a[\overline{\texttt{ff}}' : \overline{X}](\overline{Y})\{\overline{\mathbb{c}}\}\ Z \triangleright \mathcal{U} \wedge a[\overline{\texttt{ff}}' : \overline{X}](\overline{Y}) \to Z = \texttt{C.m} \quad \text{IN C}}$$

Fig. 7. Inference rules for Program, Class, and Method declarations

Contract inference for statements are presented in Figures 8 and fig:inf:assign (the latter concentrating on the assignment, which has six cases). Contract inference statements for statements have the form $\Gamma \vdash_a s : \mathbb{c} \triangleright \mathcal{U} \mid \Gamma'$, which is similar to the one for methods, with the addition of a typing environment in output, which captures the modifications performed by the assignments.

Contract inference for effectfull expressions z are presented in Figures 10. Contract inference statements for effectfull expressions have the form $\Gamma \vdash_a z : \mathbb{r}, \mathbb{c} \triangleright \mathcal{U}$, which is similar to the one for methods, with the additional future record corresponding to the expression.

Finally, contract inference for functional expressions e are presented in Figures 11. Contract inference statements for functional expressions have the form $\Gamma \vdash_a e : \overline{\mathbb{r}}$, stating that e is constructed from elements (objects or datatypes) typed with the future records $\overline{\mathbb{r}}$. Intuitively, $\overline{\mathbb{r}}$ should contain only one record for an expression, but to deal with datatypes (lists of objects for instance), we have a more accurate approximation using a set.

$$\textsc{Skip}\quad \Gamma \vdash_a \texttt{skip} : \mathbb{0} \rhd \texttt{true} \mid \Gamma$$

$$\textsc{Seq}\quad \frac{\Gamma \vdash_a s_1 : \mathbb{c}_1 \rhd \mathcal{U}_1 \mid \Gamma_1 \qquad \Gamma_1 \vdash_a s_2 : \mathbb{c}_2 \rhd \mathcal{U}_2 \mid \Gamma_2}{\Gamma \vdash_a s_1; s_2 : \mathbb{c}_1 \between \mathbb{c}_2 \rhd \mathcal{U}_1 \wedge \mathcal{U}_2 \mid \Gamma_2}$$

$$\textsc{Await}\quad \frac{\Gamma \vdash_a x : \mathbb{r} \qquad X, b \text{ fresh}}{\Gamma \vdash_a \texttt{await } x? : (a,b)^{\mathbb{w}} \rhd \mathbb{r} = b \rightsquigarrow X \mid \Gamma}$$

$$\textsc{Await-b}\quad \frac{\Gamma \vdash_a x : \mathbb{r} \qquad \overline{X}, b \text{ fresh} \qquad \mathit{class}(\mathit{types}(x)) = \mathsf{C} \qquad \mathit{fields}(\mathsf{C}) = \overline{\mathtt{f}}}{\Gamma \vdash_a [x]\,\texttt{await } y : (a,b)^{\mathbb{w}}_{\mathbb{q}} \rhd \mathbb{r} = b[\overline{\mathtt{f}} : \overline{X}] \mid \Gamma}$$

$$\textsc{If}\quad \frac{\Gamma \vdash_a e : \mathbb{r} \qquad \Gamma \vdash_a s_1 : \mathbb{c}_1 \rhd \mathcal{U}_1 \mid \Gamma_1 \qquad \Gamma \vdash_a s_2 : \mathbb{c}_2 \rhd \mathcal{U}_2 \mid \Gamma_2 \qquad \Gamma_1|_{\mathit{dom}(\Gamma)} = \Gamma_2|_{\mathit{dom}(\Gamma)}}{\Gamma \vdash_a \texttt{if } e \,\{\, s_1 \,\} \texttt{ else } \{\, s_2 \,\} : \mathbb{c}_1 + \mathbb{c}_2 \rhd \mathcal{U}_1 \wedge \mathcal{U}_2 \mid \Gamma_1|_{\mathit{dom}(\Gamma)}}$$

$$\textsc{While}\quad \frac{\Gamma \vdash_a s : \mathbb{c} \rhd \mathcal{U} \mid \Gamma'}{\Gamma \vdash_a \texttt{while } e \,\{\, s \,\} : \mathbb{c} \rhd \mathcal{U} \mid \Gamma'}$$

$$\textsc{Return}\quad \frac{\Gamma \vdash_a e : \mathbb{r} \qquad \Gamma(\texttt{destiny}) = \mathbb{s}}{\Gamma \vdash_a \texttt{return } e : \mathbb{0} \rhd \mathbb{r} = \mathbb{s} \mid \Gamma}$$

Fig. 8. Inference rules for Statements

$$\textsc{AssignVar}\quad \frac{\Gamma \vdash_a z : \mathbb{r}, \mathbb{c} \rhd \mathcal{U} \qquad x \in \mathit{dom}(\Gamma)}{\Gamma \vdash_a x = z : \mathbb{c} \rhd \mathcal{U} \mid \Gamma[x = \mathbb{r}]}$$

$$\textsc{AssignVar2}\quad \frac{\Gamma \vdash_a e : \overline{\mathbb{r}} \qquad \exists i,\ \mathbb{r}_i = \Gamma(x)}{\Gamma \vdash_a x = e : \mathbb{0} \rhd \texttt{true} \mid \Gamma}$$

$$\textsc{AssignVar3}\quad \frac{\Gamma \vdash_a e : \overline{\mathbb{r}} \qquad \forall i,\ \mathbb{r}_i \neq \Gamma(x)}{\Gamma \vdash_a x = e : \mathbb{0} \rhd \texttt{true} \mid \Gamma[x = \mathbb{r}_1]}$$

$$\textsc{AssignField}\quad \frac{\Gamma(\texttt{this}) = a[\mathtt{f} : \mathbb{r}; \overline{\mathtt{f}} : \overline{\mathbb{r}}] \qquad \mathtt{f} \notin \mathit{dom}(\Gamma) \qquad \Gamma \vdash_a z : \mathbb{r}', \mathbb{c} \rhd \mathcal{U}}{\Gamma \vdash_a \mathtt{f} = z : \mathbb{c} \rhd \mathcal{U} \wedge \mathbb{r} = \mathbb{r}' \mid \Gamma}$$

$$\textsc{AssignField2}\quad \frac{\Gamma(\texttt{this}) = a[\mathtt{f} : \mathbb{r}; \overline{\mathtt{f}} : \overline{\mathbb{r}}] \qquad \mathtt{f} \notin \mathit{dom}(\Gamma) \qquad \Gamma \vdash_a e : \overline{\mathbb{r}} \qquad \exists i,\ \mathbb{r}_i = \mathbb{r}}{\Gamma \vdash_a \mathtt{f} = e : \mathbb{0} \rhd \texttt{true} \mid \Gamma}$$

$$\textsc{AssignField3}\quad \frac{\Gamma(\texttt{this}) = a[\mathtt{f} : \mathbb{r}; \overline{\mathtt{f}} : \overline{\mathbb{r}}] \qquad \mathtt{f} \notin \mathit{dom}(\Gamma) \qquad \Gamma \vdash_a e : \overline{\mathbb{r}} \qquad \forall i,\ \mathbb{r}_i \neq \mathbb{r}}{\Gamma \vdash_a \mathtt{f} = e : \mathbb{0} \rhd \mathbb{r} = \mathbb{r}_1 \mid \Gamma}$$

Fig. 9. Inference rules for Assignment

$$\textsc{NewCog}\quad \frac{\Gamma \vdash_a \overline{e} : \overline{\mathbb{r}} \qquad a' \text{ fresh} \qquad \mathit{param}(\mathsf{C}) = \overline{\mathtt{f}}' \qquad \mathit{fields}(\mathsf{C}) = \overline{\mathtt{f}} \qquad \overline{X} \text{ fresh}}{\Gamma \vdash_a \texttt{new cog } \mathsf{C}(\overline{e}) : a'[\overline{\mathtt{f}} : \overline{X}; \overline{\mathtt{f}}' : \overline{\mathbb{r}}], \mathbb{0} \rhd \texttt{true}}$$

$$\textsc{New}\quad \frac{\Gamma \vdash_a \overline{e} : \overline{\mathbb{r}} \qquad \mathit{param}(\mathsf{C}) = \overline{\mathtt{f}}' \qquad \mathit{fields}(\mathsf{C}) = \overline{\mathtt{f}} \qquad \overline{X} \text{ fresh}}{\Gamma \vdash_a \texttt{new } \mathsf{C}(\overline{e}) : a[\overline{\mathtt{f}} : \overline{X}; \overline{\mathtt{f}}' : \overline{\mathbb{r}}], \mathbb{0} \rhd \texttt{true}}$$

$$\textsc{AInvk}\quad \frac{\Gamma \vdash_a e : \mathbb{r} \qquad \Gamma \vdash_a \overline{e} : \overline{\mathbb{s}} \qquad \mathit{class}(\mathit{types}(e)) = \mathsf{C} \qquad b, Y, \overline{Y} \text{ fresh}}{\Gamma \vdash_a e!\mathtt{m}(\overline{e}) : b \rightsquigarrow Y, \mathsf{C}!\mathtt{m}\ \mathbb{r}(\overline{\mathbb{s}}) \to Y \rhd b[\overline{\mathtt{f}} : \overline{Y}] = \mathbb{r} \wedge \mathsf{C}.\mathtt{m} \preceq \mathbb{r}(\overline{\mathbb{s}}) \to Y}$$

$$\textsc{SInvk}\quad \frac{\Gamma \vdash_a e : \mathbb{r} \qquad \Gamma \vdash_a \overline{e} : \overline{\mathbb{s}} \qquad \mathit{class}(\mathit{types}(e)) = \mathsf{C} \qquad Y \text{ fresh}}{\Gamma \vdash_a e.\mathtt{m}(\overline{e}) : a \rightsquigarrow Y, \mathsf{C}.\mathtt{m}\ \mathbb{r}(\overline{\mathbb{s}}) \to Y \rhd \mathsf{C}.\mathtt{m} \preceq \mathbb{r}(\overline{\mathbb{s}}) \to Y}$$

$$\textsc{Get}\quad \frac{\Gamma \vdash_a e : \overline{\mathbb{r}} \qquad X, b \text{ fresh}}{\Gamma \vdash_a e.\texttt{get} : X, (a,b) \rhd \mathbb{r}_1 = b \rightsquigarrow X}$$

Fig. 10. Inference rules for Effect Expressions

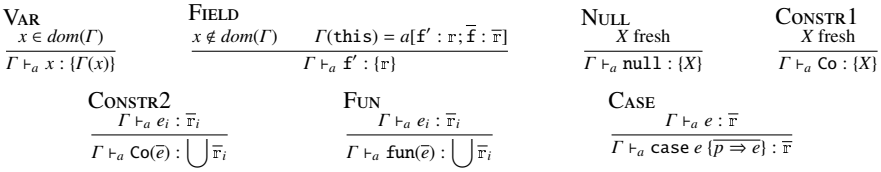

Fig. 11. Inference rules for Pure Expressions