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 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 methods in the form of behavioral types, called contracts, and (ii) an evaluator that computes a fixpoint semantics returning a finite state model 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. 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 impossible.

Deadlocks may be particularly hard to detect in systems where the basic communication operation is asynchronous and where a synchronization would explicitly occur 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`

⋆ Partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (`http://www.hats-project.eu`).

is two-tiered. At the lower level it is similar to that of JCoBox [20], which in turn generalizes the concurrency model of Creol [15] 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` [14] (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 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. There are a wide number of theories and tools for verifying their properties. However, unlike most techniques on deadlock analysis, our behavioral language handles dynamic name creation and does not require a predefined partial order.

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 would have infinitely many states (due to the creation of new objects in recursive methods). 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 models. While we lose precision, our technique is sound (in some cases, our technique may signal false positives).

We prototyped an implementation of our framework, called the SDA tool and validated it via an industrial case study. The case study is based on the Fredhopper Access Server (FAS) developed by SDL Fredhopper[4]. In particular we were able to verify 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 behavioral language for specifying contracts and the inference system for extracting contracts from `core ABS` programs. In Section 4,

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

$P ::= \overline{D}\ \overline{F}\ \overline{I}\ \overline{C}\ \{\overline{T\ x\ ;}\ \ s\,\}$	program
$T ::= \mathtt{V} \mid \mathtt{D}\langle\overline{T}\rangle \mid \mathtt{I}$	type
$D ::= \mathtt{data}\ \mathtt{D}\langle\overline{V}\rangle = \overline{\mathtt{Co}\ [\ (\overline{T})\ ]}$	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
$S ::= T\ \mathtt{m}(\overline{T\ x})$	method signature
$C ::= \mathtt{class}\ \mathtt{C}(\overline{T\ x})\ [\mathtt{implements}\ \overline{\mathtt{I}}]\ \{\,\overline{T\ x\ ;}\ \ \overline{M}\,\}$	class
$M ::= S\{\,\overline{T\ x\ ;}\ \ s\,\}$	method definition
$s ::= \mathtt{skip} \mid s\ ;\ s \mid x = z \mid \mathtt{await}\ g$	statement
$\quad \mid \mathtt{if}\ e\ \{\,s\,\}\ \mathtt{else}\ \{\,s\,\} \mid \mathtt{while}\ e\ \{\,s\,\} \mid \mathtt{return}\ e$	
$z ::= e \mid \mathtt{new}\ [\mathtt{cog}]\ \mathtt{C}\ (\overline{e}) \mid e.\mathtt{m}(\overline{e}) \mid e!\mathtt{m}(\overline{e}) \mid e.\mathtt{get}$	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

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 are discussed in Section 6 and final remarks are collected in Section 7. Due to space limitations, some technical parts are informally discussed and proofs are omitted; they can be found in the full paper.

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 [14].

In the syntax, an overlined element corresponds to any finite sequence of such element; as usual, an element between square brackets is optional. When we write $\overline{T}$ (or $\overline{V}$ or $\overline{T\ x}$ or $\overline{e}$ or $\overline{v}$) we mean a (possibly empty) sequence $T_1, \cdots, T_n$ (or, respectively, $V_1, \cdots, V_n$ or $T_1\ x_1, \cdots, T_n\ x_n$ or $e_1, \cdots, e_n$ or $v_1, \cdots, v_n$).

A program P is a list of declarations followed by a main function $\{\,\overline{T\ x\ ;}\ \ s\,\}$. Declarations include *data-types* D and *functions* F, which constitute the functional part of the language, and *interfaces* I and *classes* C, which constitute the object-oriented part of the language. A type T is the 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*. The 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, an optional sequence of type parameters $\overline{T}$ (for polymorphism), 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 be either one of the standard operations of a core imperative language or one of the operations for scheduling. Scheduling operations include `await` g, which suspends the method's execution until the argument, called a *guard*, becomes true. Guards may be a Boolean expression e that must be true in order to continue the method's execution, and a future lookup $x?$ that requires the value of x to be resolved before resuming the method's execution, or a 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 `new C` $(\overline{e})$ in the same group of the creator or an object creation `new cog 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 `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 must explicitly release control in order for another task of the same group to progress. Tasks are created by method invocations: the caller activity continues after the invocation while the called code runs as a new task. Caller and callee synchronize when the returned value of the method is strictly necessary. In order to decouple method call and returned value, `core ABS` uses futures, i.e., pointers to returned values that may not yet be available. Accesses to the future values may require waiting for the values to be returned.

The code in Figure 2 gives three different implementations of the factorial function in an hypothetical class `Math`. The function `fact_g` is the standard definition of factorial: the recursive invocation `this!fact_g(n-1)` is followed 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 as that 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.

```
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`

2.2 Restrictions of `core ABS` of the current release of SDA

In order to verify the feasibility of our techniques, in the first release of our prototype we considered a subset of `core ABS` features. Note that these restrictions have been considered in order to ease the initial development of the SDA tool. These restrictions do not jeopardize the tool's extension to the full language. Below we discuss the restrictions and, for each of them, we detail the techniques that will be used to remove them in the next release of SDA. We also notice that, notwithstanding the following restrictions, we were able to verify large commercial codes, such as a core component of FAS discussed in this paper.

Split synchronizations. `core ABS` allows synchronization primitives (`await` and `get`) to be performed long after the method invocation. Recording the associated 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. Clearly, the extension of the SDA tool with a standard alias analysis will permit the removal of this constraint.

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 synchronized with any 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`. Thus the annotation `[thread]` is needed.

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. As a result we require that: *(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 with these properties may be analyzed for deadlocks using *representatives.* Namely, a data type value is abstracted by one of its objects and a `while` loop is abstracted by its body. Note that both conditions hold in many usages of dynamic data types and iteration, particularly in the case study. The next release of SDA will permit ad-hoc annotations for `while` loops (invariants) that affect contracts generated by the inference system.

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 could lead to indeterminate behavior. In order to simplify the analysis, we constrain field assignments to keep field's record structure unchanged. For instance, if a field contains an object of group a, then that field may be only updated with objects belonging to a (and this correspondence must hold recursively with respect to the fields of objects referenced by a). 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 only be accessed by the method in which they are declared. In fact it is easy to track local changes in the inference algorithm. It is possible to be more liberal as regards fields assignments. In [12] an initial study for covering full-fledged field assignments was undertaken using so-called union types (that is, by extending the syntax of future records with a $+$ operator, as for contracts, see below) and collecting all records in the inference rule of the field assignment (and the conditional).

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 for our technique because it breaks the association of a unique abstract behavior with a method invocation. In the current release of SDA we avoid this issue by constraining codes to have interfaces implemented by at most one class. This restriction will be relaxed by admitting that methods have multiple contracts, one for every possible class implementation of the arguments, and return values of methods are unions of records. In turn, method invocations yield unions of contracts, according to the possible instantiations of their arguments.

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. Currently, the SDA tool cannot deal with recursive structures, because the semi-unification process associates each object with a finite tree structure. In this way, it is not possible to capture circular definitions, such as the recursive ones. This restriction will be removed in the next release of SDA by admitting the association of *regular terms* [5] with objects in the semi-unification process.

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, ..., and *group names* a, b, ... 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 $\mathbb{r}$ encode the values of expressions in contracts. A record may be one of the following: an empty record $\_$, which corresponds to primitive types; a record name X, which represents a place-holder for a value and can be instantiated by substitutions; $a[\overline{\mathtt{f}} : \overline{\mathbb{r}}]$ that defines an object with its group name a, and $a \rightsquigarrow \mathbb{r}$ which specifies that accessing $\mathbb{r}$ requires control of the group a (and that the control is to be released once the method has been evaluated). Note that the future record $a \rightsquigarrow \mathbb{r}$ is associated with method invocations: a is the group of the object on which the method is invoked.

Contracts $\mathbb{c}$ collect the method invocations and the group dependencies inside statements. Apart from $\mathtt{0}$, (a, a'), and $(a, a')^{\mathtt{w}}$ that respectively represent the empty behavior, the dependency pairs due to a `get` and an `await` operation, the other basic contracts deal with method invocations. The contract $\mathtt{C.m}\,\mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}'$ models synchronous method invocations, while $\mathtt{C!m}\,\mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}'$ models asynchronous invocations. This latter contract may be followed by a `get` – the suffix "$\centerdot (a, a')$" –, or followed by an `await` – the suffix "$\centerdot (a, a')^{\mathtt{w}}$". Composite contracts define the sequential composition $\mathbb{c} \fatsemi \mathbb{c}'$ and conditionals $\mathbb{c} + \mathbb{c}'$.

Finally, our tool uses constraints $\mathcal{U}$ that are 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 $\mathcal{U}'$.

Contracts are extracted from `core ABS` programs by means of an inference algorithm. Figures 3 and 4 illustrate a (relevant) subset of the rules; the other ones are omitted to lighten our presentation. The following auxiliary operators are used: *fields*(`C`) and *param*(`C`) return the sequence of fields and parameters of a class

$$\textsc{Var}\quad \frac{x \in \mathit{dom}(\Gamma)}{\Gamma \vdash_a x : \Gamma(x)}$$

$$\textsc{Field}\quad \frac{x \notin \mathit{dom}(\Gamma) \qquad \Gamma(\mathtt{this}) = a[\mathtt{f}' : \mathbb{r}; \overline{\mathtt{f}} : \overline{\mathbb{r}}]}{\Gamma \vdash_a \mathtt{f}' : \mathbb{r}}$$

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 3. Contract inference for expressions and statements

`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 e have the form $\Gamma \vdash_a e : \mathbb{r}$, where Γ is a typing context mapping variables to their records, and methods to their signatures; a is the group name of the object executing 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. The rule NewCOG creates a new group name that is returned in the record of the expression, while New uses the name of the group 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`, the inference system in our tool explicitly

(Method)

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

(Class)

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

Fig. 4. Contract rules of method and class declarations

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, acting as if `init` and `run` are always 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, and Γ' is the environment of the method after the execution of the statement. The environment may change because of local variable updates. Rule Await deals with the `await` synchronization when applied to a simple future lookup $x?$, returning a dependency $(a, b)^{\mathtt{w}}$. In order to correctly associate dependencies with each synchronization, we assume statements of the form `await` $(?x_1 \wedge ?x_2)$ to be decomposed into `await` $?x_1$ `;` `await` $?x_2$. Rule AssignVar manages assignments to local variables of methods and is the only rule that changes the environment. This rule must be compared with AssignField, which deals with assignment to 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 to be the same as that of the field. Rule Return constrains 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 binary operator $\between$ on contracts to manage accumulations of dependencies in sequence. The operator $\between$ is defined case-by-case. For example

$$\mathbb{c} \fatsemi \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{r}' \between \mathbb{c}' = \begin{cases} \mathbb{c} \fatsemi \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{r}' \centerdot (a, b) \fatsemi \mathbb{c}'' & \text{if } \mathbb{c}' = (a, b) \fatsemi \mathbb{c}'' \\ \mathbb{c} \fatsemi \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{r}' \centerdot (a, b)^{\mathtt{w}} \fatsemi \mathbb{c}'' & \text{if } \mathbb{c}' = (a, b)^{\mathtt{w}} \fatsemi \mathbb{c}'' \\ \mathbb{c} \fatsemi \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{r}' \fatsemi \mathbb{c}' & \text{otherwise} \end{cases}$$

The rules for method and class declarations are defined in Figure 4. In Method, in order to derive the method contract of $\mathtt{T}\ \mathtt{m}\ (\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{s\}$, 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. The rule Class yields an *abstract class table* that associates a method contract with every method name. It is this abstract class table that is used by our analyzer in Section 4.

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

– `fact_g` has contract $a[\,](\_)\ \{\mathtt{0}{+}\mathtt{Math!fact\_g}\ a[\,](\_) \to \_ \centerdot (a, a)\}\ \_$. The name a in the header refers to the group name associated with `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() ; Fut<Int> x = z!fact_g(n-1)`, we obtain the same contract as above because the new object is in the same group as `this`.

– `fact_ag` has contract $a[\,](\_)\ \{\mathtt{0} + \mathtt{Math!fact\_ag}\ a[\,](\_) \rightarrow \_\,\boldsymbol{.}\,(a, a)^{\mathtt{w}}\}\ \_$. 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` synchronization.

– `fact_nc` has contract $a[\,](\_)\ \{\mathtt{0} + \mathtt{Math!fact\_nc}\ b[\,](\_) \rightarrow \_\,\boldsymbol{.}\,(a, b)\}\ \_$. 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.

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

This result is proved in a standard way by (1) defining a type system for contracts 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 [14] (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 [13] $\mathbb{r}(\overline{\mathbb{r}}) \rightarrow \mathbb{s} \preceq \mathbb{r}'(\overline{\mathbb{r}}') \rightarrow \mathbb{s}'$ by binding constraints of the form $\mathbb{r}(\overline{\mathbb{r}}) \rightarrow \mathbb{s} = \mathtt{C.m}$ (rule Method) with constraints of the form $\mathtt{C.m} \preceq \mathbb{r}'(\overline{\mathbb{r}}') \rightarrow \mathbb{s}'$ (rules AInvk and SInvk). It is well-known that solving these constraints is undecidable in general [17]. Therefore, it is to be expected that the algorithm loops indefinitely in some cases, which are defined in very ad-hoc ways. In our various tests, we never reached this limitation of our approach.

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 [11]), 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; then it 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 classical example is the model of the *recursive* method contract (of `Math.fact_ng`)

$$\texttt{Math.fact\_nc}\ a[\,](\_)\ \{\texttt{0} + \texttt{Math!fact\_nc}\ b[\,](\_) \to \_\bullet(a,b)\}\ \_$$

In order to circumvent this issue and to get a decision on 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 resorting to 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-th 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 saturated state $[\![(a_1,a_2),\cdots,(a_{n-1},a_n),(a_2,a_2)]\!]$. This state contains a circular dependency – the pair (a_2,a_2) – revealing a potential deadlock in the corresponding program. Actually, 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 (a simplified version of the algorithm may be found in [9], see also Section 6). 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 with every method a function from tuples of group names to *pairs of lams*. The need for using pairs of lams, let them be $\langle\mathcal{W},\mathcal{W}'\rangle$, is illustrated by means of an example. Consider the contract $\mathbb{c} = \texttt{C!m}\ b[\ ](\ ) \to \_\bullet(a,b)$. 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 $\texttt{D!n}\ b[\ ](\ ) \to \_$ (without any `get` or `await` synchronization), then the invocation $\texttt{C!m}\ b[\ ](\ ) \to \_$ does not contribute to the current state with other pairs. However it is possible that $\texttt{D!n}\ b[\ ](\ ) \to \_$ introduces dependency pairs that affect the *future states* and that have nothing to do with (a,b). The same arguments apply in the case where `D!n` is a set of states: future dependency pairs are added according to what prescribed by the model of `D!n`. Therefore, the dichotomy between present and future states allows us to augment the precision of our (compositional) abstract semantics. We notice that this dichotomy 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 $\mathcal{W}_{main}\cup\mathcal{W}'_{main}$ – in this case, futures are simply additional states to the current ones.

Back to the abstract class table, it is computed starting from the first approximant, which associates the function $\lambda\widetilde{a_{\texttt{C.m}}}.\langle \texttt{0}, \texttt{0}\rangle$ with 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, the lam of the main function $\{\, \overline{T\ x\ ;}\ \ s\,\}$ is computed. Let $\langle \mathcal{W}_{main}, \mathcal{W}'_{main}\rangle$ be such lam. The input program is then 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 abstract class table of the 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`	$\lambda a.\langle \texttt{0}, \texttt{0}\rangle$	$\lambda a.\langle [\![(a,a)]\!], \texttt{0}\rangle$	$\lambda a.\langle [\![(a,a)]\!], \texttt{0}\rangle$
`Math.fact_ag`	$\lambda a.\langle \texttt{0}, \texttt{0}\rangle$	$\lambda a.\langle [\![(a,a)^{\texttt{w}}]\!], \texttt{0}\rangle$	$\lambda a.\langle [\![(a,a)^{\texttt{w}}]\!], \texttt{0}\rangle$
`Math.fact_nc`	$\lambda a.\langle \texttt{0}, \texttt{0}\rangle$	$\lambda a.\langle [\![(a,b)]\!], \texttt{0}\rangle$	$\lambda a.\langle [\![(a,c),(c,d)]\!], \texttt{0}\rangle$

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`. If we decide to stop at the third iteration and saturate, we get $\lambda a.\langle [\![(a,c),(c,c),(c,d)]\!], \texttt{0}\rangle$, which contains a circularity. As we said before, this circularity is a false positive.

Note that saturation might even start at the first approximant (where every method is $\lambda a.\langle \texttt{0}, \texttt{0}\rangle$). In this case, for `Math.fact_g` and `Math.fact_ag`, we get the same answer and the same pair of lams as the above third approximant. For `Math.fact_nc` we get $\lambda a.\langle [\![(a,b),(b,b)]\!], \texttt{0}\rangle$, which contains a circularity. In general, it is possible to augment precision by delaying saturation. Consider the following abstract class table:

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

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 its application to the case study

`ABS` (and, therefore, `core ABS`) comes with a suite [25] that offers a compilation framework, a set of tools to analyze the code, an Eclipse IDE plugin and Emacs mode for the language. We extended this suite with an implementation of our static deadlock analysis tool (SDA tool), available at `http://cs.unibo.it/~laneve/deadlock`. The SDA tool is built upon the abstract syntax tree (AST) of the ABS type checker. We can therefore exploit the type information stored in every node of the tree. This simplifies the implementation of several contract inference rules. The SDA tool is structured in three modules.

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 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.
2. *Constraint Solving* is done by a generic semi-unification solver implemented in Java, following the algorithm defined in [13]. The implementation of that solver is available at `http://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 contract (because states become larger and larger as the analysis progresses). At each iteration of the analysis, a number of fresh group names is created and the states are updated according to what is prescribed by the contract. A basic operation of the analyzer is the renaming, which is used when computing every approximant. At each iteration, the tool checks whether a fixpoint has been reached. Saturation starts when the number of iterations reaches a maximum value (that may be customized by the user). In this case, since the precision of the algorithm degrades, the tool signals that the answer may be imprecise.

5.1 Simple experiments

The SDA tool has been tested on a number of medium-size programs written for benchmarking purposes by ABS programmers. The programs may be found on the tool website; some of them (those with suffix `Mod`) required modifications to remove recursive object structures. The following table reports our experiments: for every program we display the number of lines, whether the analysis has reported a deadlock (`D`) or not (✓), and the time required for the analysis. With regards to time, we only report the time required by the contract inference system and the contract analysis when they run on a QuadCore 2.4GHz and Gentoo (Kernel 3.4.9):

program	lines	result	time
`PeerToPeer`	185	✓	0.474 sec
`BoundedBuffer`	103	✓	0.353 sec
`PingPongMod`	61	✓	0.046 sec
`MultiPingPongMod`	88	`D`	0.109 sec

5.2 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 to provide 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 single *SyncServer* module and one *SyncClient* module for each live environment. In turn, the *SyncServer* determines the schedule of replications, as well as their content, while a *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 with read/write accesses to 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 their update is deemed safe. The *SyncServer* uses a *Coordinator* object to determine the safe state in which the *Snapshot* can be refreshed.

5.3 The application of SDA to FAS

As the *Replication Protocol* is a program with multiple threads interacting concurrently, there are risks of deadlock. In order to be able to to apply the SDA tool to the case study, we first made 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). In total we 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. In total we modified three circular object structures to be non recursive.

Finally, we have annotated every `await` statement on boolean guards with a 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 to 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. We ran it with number of iterations `1` and, within 40 seconds, we got the answer

```
### LOCK INFORMATION RESULTED BY THE ANALYSIS ###
  Saturation: true
  Deadlock in Main: false
```

In order to test the performance of SDA, we have also run it with other iteration values (which are not necessary for the functional analysis, in this case). The following table summarizes the results of our experiments:

Replication Protocol	time
`Iteration 1`	39.783 sec
`Iteration 2`	60.582 sec
`Iteration 3`	341.10 sec

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 is the reason why, in the above table, increasing the number of iterations (from `2` to `3`) causes the runtime to increase by a factor of 6. We remark that in most cases, the precision of the SDA tool does not enhance at iterations higher than `1`.

6 Related works

A preliminary theoretical study was undertaken in [10], where (*i*) the considered language is a functional subset of `core ABS`; (*ii*) contracts are not inferred, they are provided by the programmer and type-checked; (*iii*) the deadlock analysis is less precise because it is not iterated as in this contribution, but stops at the first approximant, and (*iv*), more importantly, models of methods are not pairs of lams, which led it to discard dependencies (thereby causing the analysis, in some cases, to yield false negatives).

The proposals in the literature that statically analyze deadlocks are largely based on types. In [1, 3, 7, 23] 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 compute 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. In [18, 21, 22], Kobayashi and his colleagues use a very powerful technique, since they do not commit to any predefined partial order of locks and apply to codes with dynamic structures. However their concurrency models are different from that of `ABS` and a precise comparison is a matter for future work. Type-based deadlock analysis has also been studied in [19]. In this contribution, types define objects' states and can express acceptability of messages. The exchange of messages modifies the state of the objects. In this context, a deadlock is avoided by setting an ordering on types. With respect to our technique, [19] uses a deadlock prevention approach, rather than detection, and no inference system for types is provided. A number of model-theoretic techniques for deadlock analysis have also been defined.To mention one contribution (another one is [6], see below), in [4], circular dependencies among processes are detected as erroneous configurations, but dynamic creation of names is not treated.

Works that specifically tackle the problem of deadlocks for languages with the same concurrency model as that of `core ABS` are the following: [24] defines an approach for deadlock prevention (as opposed to our deadlock detection) in SCOOP, an Eiffel-based concurrent language. Different from our approach, they annotate classes with the used *processors* (the analogue of groups in ABS), while this information is inferred by our technique. Moreover each method exposes preconditions representing required lock ordering of processors (processors obeys an order in which to take locks), and this information must be provided by the programmer. [6] studies a Petri net based analysis, reducing deadlock detection to a reachability problem in Petri nets. This technique is more precise in that it is thread based and not just object based. Since the model is finite, this contribution does not address the feature of object creation and it is not clear how to scale the technique. We plan to extend our analysis in order to consider finer-grained thread dependencies instead of just object dependencies. [16] offers a design pattern methodology for CoJava to obtain deadlock-free programs. CoJava, a Java dialect where data-races and data-based deadlocks are avoided by the type system, prevents threads from sharing mutable data. Deadlocks are excluded by a programming style based on ownership types and *promise* (i.e. future) objects. The main differences with our technique are (i) the needed information must be provided by the programmer, (ii) deadlock freedom is obtained through ordering and timeouts, and (iii) no guarantee of deadlock freedom is provided by the system.

The work by Flores-Montoya *et al.* [8] and the corresponding DECO prototype deserve a separate discussion. They perform deadlock analysis on (a subset of) `core ABS` with a point-to analysis technique that returns a dependency graph. Then, in a clever way (by means of a may-happen-in-parallel analysis), unfeasible cycles in the dependency graph are discarded. The technique relies on an abstract evaluation of the code; therefore no inference system for extracting relevant informations is used. For this reason, the DECO tool does not manifest limitations of the current version of SDA, such as recursive object structures. As regards performance, DECO and SDA are comparable on small/mid-size programs (codes in Section 5.1). In case of the FAS module, DECO provides an answer in a bit more than 4 seconds. As regards the design, DECO is a monolithic code written in Prolog. On the contrary, SDA is a highly modular Java code (see Section 5). Every module may be replaced by another; for instance one may rewrite the inference system for another language and plug it easily in the tool, or one may use a different/refined contract analysis algorithm (see Conclusions).

7 Conclusions

We have developed a technique for statically detecting deadlocks in `core ABS` and discussed an industrial case study. The technique uses (i) an inference algorithm to extract abstract descriptions of methods, called contracts, and (ii) an evaluator of contracts, which computes an over-approximated fixpoint semantics.

This study can be extended in several directions. As regards the prototype, in the next release, we intend to remove most of the restrictions, as discussed in Section 2.2, since they have been considered only to ease the initial version. The next release of SDA will also provide indications about *how* deadlocks have been produced by pointing out the elements in the code that generated the detected circular dependencies. This way, the programmer will be able to check whether or not the detected circularities are actual deadlocks, fix the problem in case it is a verified deadlock, or be assured that his program is deadlock-free.

The current SDA tool is also able to capture (a form of) *livelock*, namely when several processes are continuously releasing and acquiring a set of group locks in a circular way. However, the theoretical development of this issue is at an early stage and we will extend the tool when the theory becomes more stable. SDA, being modular, may be integrated with other analysis techniques. In particular, we are prototyping the technique discussed in [11], which extends the theory of permutations to the contracts discussed in this paper. This technique provides a deadlock analysis that is complementary to the one discussed here. In the sense that there are programs that are false-positive in one technique and deadlock-free in the other, and conversely. Once this work is carried out, we will have an SDA tool with augmented precision.

Acknowledgment We wish to thank Stephan Schroevers for proof reading this paper and the anonymous referees for useful suggestions and comments.

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. M. Coppo. Type Inference with Recursive Type Equations. In *Proc. FoSSaCS*, volume 2030 of *LNCS*, pages 184–198. Springer, 2001.
6. F. de Boer, M. Bravetti, I. Grabe, M. Lee, M. Steffen, and G. Zavattaro. A petri net based analysis of deadlocks for active objects and futures. In *Proc. of Formal Aspects of Component Software - 9th International Workshop, FACS 2012*, volume 7684 of *Lecture Notes in Computer Science*, pages 110–127. Springer, 2012.
7. 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.
8. A. Flores-Montoya, E. Albert, and S. Genaim. May-happen-in-parallel based deadlock analysis for concurrent objects. In *Proc. FORTE/FMOODS 2013*, LNCS. Springer, 2013.
9. E. Giachino and C. Laneve. Analysis of deadlocks in object groups. In *Proc. FMOODS/FORTE*, volume 6722 of *LNCS*, pages 168–182. Springer, 2011.
10. 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.
11. E. Giachino and C. Laneve. A beginner's guide to the deadLock Analysis Model. In *TGC*, Lecture Notes in Computer Science. Springer-Verlag, 2013.
12. E. Giachino and T. A. Lascu. Lock Analysis for an Asynchronous Object Calculus. In *Proc. 13th ICTCS*, 2012.
13. F. Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, Apr. 1993.
14. 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.
15. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and System Modeling*, 6(1):35–58, Mar. 2007.
16. E. Kerfoot, S. McKeever, and F. Torshizi. Deadlock freedom through object ownership. In T. Wrigstad, editor, *5rd International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO), in conjunction with ECOOP 2009*, July 2009.
17. A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. *Inf. Comput.*, 102(1):83–101, 1993.
18. 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.

19. F. Puntigam and C. Peter. Types for active objects with static deadlock prevention. *Fundam. Inform.*, 48(4):315–341, 2001.
20. 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.
21. 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.
22. 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.
23. 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.
24. S. West, S. Nanz, and B. Meyer. A modular scheme for deadlock prevention in an object-oriented programming model. In *ICFEM*, pages 597–612, 2010.
25. 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.