Project N°:	**FP7-610582**
Project Acronym:	**ENVISAGE**
Project Title:	**Engineering Virtualized Services**
Instrument:	**Collaborative Project**
Scheme:	**Information & Communication Technologies**

Deliverable D3.2
Verification

Date of document: T18

Start date of the project: **1st October 2013** Duration: **36 months**

Organisation name of lead contractor for this deliverable: **TUD**

Final version

STREP Project supported by the 7th Framework Programme of the EC		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Executive Summary:
Verification

This document summarises deliverable D3.2 of project FP7-610582 (Envisage), a Collaborative Project supported by the 7th Framework Programme of the EC within the Information & Communication Technologies scheme. Full information on this project is available online at `http://www.envisage-project.eu`.

This report presents static analysis methods which can be used to formally verify the behavior of an abstract system model (Task T1.1) against contracts derived from behavioral interfaces (Task T2.1).

The following aspects are reported:

1. the sound compositional reasoning system in dynamic logic for ABS
2. the implementation of the deductive verification tool `KeY-ABS`
3. the application of `KeY-ABS` on unbounded systems
4. the tool `DF4ABS` for deadlock detection

List of Authors

Crystal Chang Din (TUD)
Richard Bubel (TUD)
Reiner Hähnle (TUD)
Elena Giachino (BOL)
Cosimo Laneve (BOL)
Michael Lienhardt (BOL)

Contents

Chapter 1

Introduction

According to the Envisage Description of Work, the Deliverable 3.2

- develops static analysis methods which can be used to formally verify compliance of system models with their contracts. In other words, we ensure that any possible behavior of an abstract system model (Task T1.1) complies with requirements derived from behavioral interfaces (Task T2.1).

In ABS[1], as developed in Task T1.1, two different kinds of concurrency are supported depending on whether two objects belong to the same or to different *concurrent object groups* (COGs). Within a COG several threads might exist, but only one of these threads (and hence one object) can be active at any time. Another thread can only take over when the current active thread explicitly releases control. In other words, ABS realizes *cooperative scheduling* within a COG. All interleaving points occur syntactically explicit in an ABS program.

While one COG represents a single processor with task switching and shared memory, two different COGs run actually in parallel and are separated by a network. As a consequence, objects within the same COG may communicate either by asynchronous or by synchronous method invocation, while objects living on different COGs *must* communicate with asynchronous method invocation and message passing. Technically, the decoupling of method invocation and the returned value is realised using future variables [4], which are pointers to values that may be not available yet. Clearly, accessing values of future variables may require waiting for the value to be returned.

Task T2.1 introduces two notions of behavioral interface for the basic virtualized services as developed in Task T1.1. These interfaces are mostly used to verify the interoperability obligations between the different (concurrent) parties that compose a virtual system. One type of behavioral interface is based on communication histories. These histories abstractly capture the system state at any point in time. Partial correctness properties of a system may thereby be specified by the set of finite initial segments of its communication histories. A history invariant is a predicate over the communication history, which holds for all finite sequences in the (prefix-closed) set of possible histories. Such invariants express safety properties. Another type of behavioral interface is called *lams*, which are sequences of basic terms recording the method invocations and the synchronisations between calling and called methods. Task T2.1 provides the formalisation of the obligations of components of ABS systems that guarantee never-ending waitings for returned values. This property, which turns out to be a consequence of deadlock freedom, is enforced by associating behavioral interfaces to method definitions.

In this Deliverable, we describe two tools. Each tool can formally verify compliance of system models with their contracts derived from each type of behavioral interface, respectively. Chapters 2-4 relate to the history-based behavioral interface and Chapter 5 relates to the behavioral interface using *lams*.

In Chapter 2 a sound compositional reasoning system for ABS is formulated within dynamic logic. This reasoning framework establishes an invariant over the global history from invariants over the local histories of each object. In Chapter 3 we present the deductive verification tool `KeY-ABS` which realises the dynamic

[1]The ABS language used in this Task excludes the modeling of variability in software product line engineering [3].

logic reasoning system introduced in the previous chapter. According to the Envisage Description of Work, the design of `KeY-ABS` concerns the following: (i) symbolic execution is used as the underlying key technology for state exploration, (ii) generalized contracts provide compositionality of verification and (iii) there is an explicit representation of symbolic states. Besides design issues we also discuss the usage of `KeY-ABS` and its application in Chapter 4, in which we show that `KeY-ABS` can handle unbounded and complex concurrent ABS systems. In Chapter 5 we present the `DF4ABS` tool, which (i) automatically infers the *lams* associated to each method of an ABS program and (ii) detects possible deadlocks by analysing those *lams*. *Lams* provide useful information for inter-cog synchronisation dependencies which are analysed in this task in order to discover possibly harmful circular waits for resources.

Currently we are investigating the relation between the two types of behavioral interfaces. The ultimate goal is to bridge the gap between the two techniques by translating the inference system for deadlock analysis developed at BOL into a dynamic logic proof system for KeY and reformulate *lams* as suitable invariants on communication histories. This work is planned to be done within the scope of Task T3.4, Hybrid Analysis.

1.1 List of Papers Comprising Deliverable D3.2

This section lists all the papers that this deliverable comprises, indicates where they were published, and explains how each paper is related to the main text of this deliverable. The deliverable contains either extended abstracts of the papers or the parts that are relevant for the Envisage project. The full papers are made available in the appendix of this deliverable and on the Envisage web site at the url `http://www.envisage-project.eu/` (select "Dissemination"). Direct links are also provided for each paper listed below.

Paper 1: A Sound Compositional Reasoning System in Dynamic Logic This paper presents a model for ABS asynchronously communicating objects, where return values from method calls are handled by futures. The model facilitates invariant specifications over the locally visible communication history of each object. Compositional reasoning is supported and proved sound, as each object may be specified and verified independently of its environment. A compositional proof system for ABS is presented, formulated within dynamic logic.

The paper was written by Crystal Chang Din and Olaf Owe and was published in the Journal of Formal Aspects of Computing (FAC) 2014.

Download the paper at `http://dx.doi.org/10.1007/s00165-014-0322-y`.

Paper 2: The Deductive Verification Tool: KeY-ABS We present `KeY-ABS`, a tool for deductive verification of concurrent and distributed programs written in ABS. `KeY-ABS` allows to verify data dependent and history-based functional properties of ABS models. In this paper we give a glimpse of system workflow, tool architecture, and the usage of `KeY-ABS`. In addition, we briefly present the syntax, semantics and calculus of KeY-ABS Dynamic Logic (ABSDL). The system is available for download at `http://www.envisage-project.eu/?page_id=1558`.

The paper was written by Crystal Chang Din, Richard Bubel and Reiner Hähnle and was submitted to an international conference.

Paper 3: The NoC Verification Case Study with KeY-ABS We present a case study on scalable formal verification of distributed systems that involves a formal model of a Network-on-Chip (NoC) packet switching platform. We provide an executable model of a generic $m \times n$ mesh chip with unbounded number of packets, the formal specification of certain safety properties, and formal proofs that the model fulfils these properties. The modeling has been done in ABS. Our paper shows that this is indeed possible and so advances the state-of-art verification of NoC systems. It also demonstrates that deductive verification is a viable alternative to model checking for the verification of unbounded concurrent systems that can effectively deal with state explosion.

The paper was written by Crystal Chang Din, S. Lizeth Tapia Tarifa, Reiner Hähnle, and Einar Broch Johnsen and was submitted to an international conference.

Paper 4: Deadlock Analysis A framework for statically detecting deadlocks in ABS is presented. The basic component of the framework is a front-end inference algorithm that extracts abstract behavioral descriptions of methods, called contracts, which retain resource dependency information. This component is integrated with a number of possible different back-ends that analyse contracts and derive deadlock information. As a proof-of-concept, two such back-ends are discussed: (i) an evaluator that computes a fixpoint semantics and (ii) an evaluator using abstract model checking.

The paper was written by Elena Giachino, Cosimo Laneve and Michael Lienhardt and was published in the Journal of Software and Systems Modeling (SoSyM) 2015.

Download the paper at `http://dx.doi.org/10.1007/s10270-014-0444-y`.

Chapter 2

A Sound Compositional Reasoning System in Dynamic Logic

2.1 Introduction

In Deliverable D2.1 [5] Section 3.2 we introduced the representation of behavioral interfaces using communication histories. We formalised object communication by an event-based operational semantics, capturing shared futures. Since message passing in ABS is asynchronous, we consider separate events for method invocation, reacting upon a method call, resolving a future, fetching the value of a future, and object creation. Each event is visible to only one object. Consequently, the local histories of two different objects share no common events, and invariants based on communication histories can be established independently for each object. The global history of the whole system is formed by the composition of the local history of each instance of the class, and a global invariant can be obtained as a conjunction of the class invariants for all objects in the system, adding well-formedness [7] of the global history. The well-formedness serves as a connection between the local histories, relating events with the same future to each other.

In this chapter we continue the work and develop a history-based *dynamic logic* proof system for class verification of ABS, facilitating independent reasoning about each class. A verified *class invariant* can be instantiated to each object of that class, resulting in an invariant over the local history of the object. The compositionality of this dynamic logic proof system is proved sound.

2.2 The ABS Dynamic Logic

Specification and verification of ABS models is done in ABS dynamic logic (ABSDL). ABSDL is a typed first-order logic plus a box modality: For an ABS program s and ABSDL formulae ψ and ϕ, the formula $\psi \rightarrow [s]\phi$ expresses: If the execution of a program s starts in a state where the assertion ψ holds and the program terminates normally, then the assertion ϕ holds in the final state. Hence, $[\cdot]$ acts as a partial correctness modality operator.

Verification of an ABSDL formula proceeds by symbolic execution of s, where state modifications are handled by the *update* mechanism [1]. A dynamic logic formula $[v = e; s]\phi$, i.e., where an assignment is the first statement, reduces to $\{v := e\}[s]\phi$, in which $\{v := e\}$ is an update and s is the remaining program. Updates can only be *applied* on formulas or terms without programs, which means that updates on a formula $[s]\phi$ are accumulated and their application is *delayed* until the symbolic execution of s is complete. Update application $\{v := t\}e$, on an expression e, evaluates to the substitution e^v_t, replacing all free occurrences of v in e by t. There are operations for sequential as well as parallel composition of updates, i.e., $\{v_1 := e_1 || ... || v_n := e_n\}$.

2.3 Compositional Reasoning

The concurrency model of ABS admits a proof system that is modular and permits to reduce correctness of concurrent programs to reasoning about sequential ones [2, 6]. We must prove a class invariant I_C for class C is established by the initialization code, satisfied at any process release points and maintained by all method definitions in class C, assuming well-formedness of the local history. Any execution between where the class invariants are verified is sequential.

For a method definition $m(\overline{x})\{s;\ \texttt{return}\ e\}$ in C where $\overline{x}$ is a list of method parameters, s the method body, and e the return data, this amounts to a proof of the sequent:

$$\begin{aligned}\vdash \mathit{wf}(\mathcal{H}) \wedge I_C \Rightarrow [&\mathcal{H} = \mathcal{H} \cdot \langle \mathsf{caller} \twoheadrightarrow \texttt{this}, \mathsf{destiny}, m, \overline{x}\rangle;\\ &s;\ \mathcal{H} = \mathcal{H} \cdot \langle \leftarrow \texttt{this}, \mathsf{destiny}, m, e\rangle](\mathit{wf}(\mathcal{H}) \Rightarrow I_C)\end{aligned}$$

where $\mathit{wf}(\mathcal{H})$ expresses the well-formedness of local history $\mathcal{H}$. The method body is extended with an assignment statement of history extension, $\mathcal{H} = \mathcal{H} \cdot \langle \mathsf{caller} \twoheadrightarrow \texttt{this}, \mathsf{destiny}, m, \overline{x}\rangle$, capturing the starting point of method execution. The variable destiny contained in the appended *invocation reaction event* is the future variable generated for this current method execution. The `return` statement is treated as another history extension, $\mathcal{H} = \mathcal{H} \cdot \langle \leftarrow \texttt{this}, \mathsf{destiny}, m, e\rangle$, in which the *future event* captures the termination of method execution. The definition of communication events can be found in D2.1 [5] Section 3.2.

The deductive component of our reasoning system is an axiomatization of the operational semantics of ABS in the form of a sequent calculus. We define proof rule for each ABS statement. For example, the rule for method invocation is:

$$\mathsf{invoc}\ \frac{\vdash \forall u.\, \{\mathit{fr} := u \mid\mid \mathcal{H} := \mathcal{H} \cdot \langle \texttt{this} \to v, u, m, \overline{e}\rangle\}\ [s]\phi}{\vdash [\mathit{fr} = v!m(\overline{e});\ s]\phi}$$

When invoking a method, the update in the premise of rule invoc captures the generation of a fresh future identity u and the history extension, in which the invocation of method m on object v is captured in an *invocation event* $\langle \texttt{this} \to v, u, m, \overline{e}\rangle$. Similarly, an *object creation event* containing a fresh object identity is appended to the history upon symbolically executing an object creation statement. The details of this proof rule can be found in Appendix A.

The proof rule for modular reasoning is formulated below:

$$\mathsf{composition}\ \frac{\vdash I_C(h),\ \text{for each C in } \overline{Cl}}{\overline{Cl} \vdash \mathit{wf}(h) \wedge \bigwedge_{(o:C(\overline{e}))\in ob(h)} I_{o:C(\overline{e})}(h)}$$

where we show that the obtained global invariant can be proved (in the conclusion) if all the class invariants belong to the system are provable (in the premise). The variable $\overline{Cl}$ is the set of classes in the system, $\mathit{wf}(h)$ the well-formedness of global history h, ob(h) the set of objects generated within the history h, $o : C(\overline{e})$ the instance of class C, and $I_{o:C(\overline{e})}(h)$ the invariant for object $o : C(\overline{e})$. This rule allows to prove global safety properties of the system using local rules and symbolic execution. In Appendix A we provide the detailed soundness proof for our compositional reasoning system and show a verified ABS concurrent program is correct for all possible scheduling.

Chapter 3

The Deductive Verification Tool: KeY-ABS

3.1 Introduction

`KeY-ABS` is a deductive verification system for the concurrent modelling language ABS. It is based on the KeY theorem prover [1] and realises the reasoning system given in Chapter 2. `KeY-ABS` provides an interactive theorem proving environment and allows one to prove properties of object-oriented and concurrent ABS models. The deductive component of `KeY-ABS` is an axiomatization of the operational semantics of ABS in the form of a sequent calculus for first-order dynamic logic for ABS (ABSDL). The rules of the calculus that axiomatize program formulae define a symbolic execution engine for ABS. The system provides heuristics and proof strategies that automate large parts of proof construction. For example, first-order reasoning, arithmetic simplification, symbolic state simplification, and symbolic execution of loop- and recursive-freef programs are performed mostly automatically. The remaining user input typically consists of universal and existential quantifier instantiations.

In contrast to model checking, `KeY-ABS` allows to verify complex functional properties of systems with unbounded size, see Chapter 4. `KeY-ABS` itself consists of around 11,000 lines of Java code (`KeY-ABS` + reused parts of KeY: ca. 100,000 lines in total). The rule base consists of around 10,000 lines written in KeY's *taclet* rule description language [1]. At `http://www.envisage-project.eu/?page_id=1558` the `KeY-ABS` tool can be downloaded.

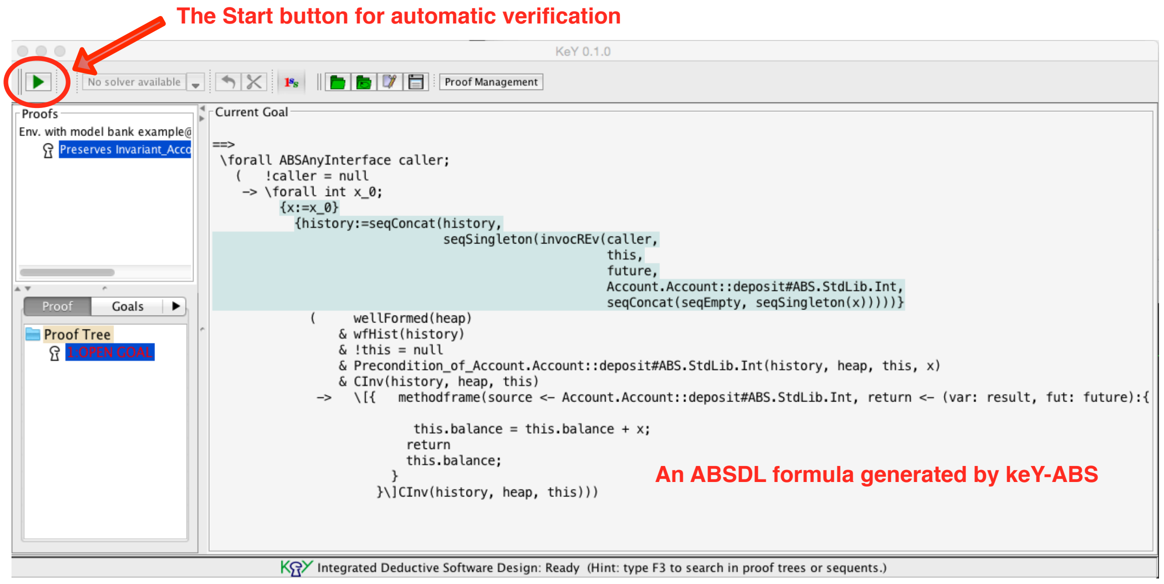
In Appendix B we provide a more detailed explanation of `KeY-ABS`. Here we give a glimpse of system workflow, tool architecture and the usage of `KeY-ABS`.

3.2 System Workflow

The input files to `KeY-ABS` comprise (i) an *.abs* file containing ABS programs and (ii) a *.key* file containing the class invariants, functions, predicates and specific proof rules required for a particular verification case. Given these input files, `KeY-ABS` opens a proof obligation selection dialogue that lets one choose a target method implementation. From the selection the proof obligation generator creates an ABSDL formula. By clicking on the **Start** button (see Fig. 3.1) the verifier will try to automatically prove the generated formula. A positive outcome shows that the target method preserves the specified class invariants. In the case that a subgoal cannot be proved automatically, the user is able to interact with the verifier to choose proof strategies and proof rules manually. The reason for a formula to be unprovable might very well be that the target method implementation does not preserve one of the class invariants, that the specified invariants are too weak/too strong or that additional proof rules are required.

3.3 KeY-ABS Architecture

Fig. 3.2 depicts the architecture of the `KeY-ABS` system. `KeY-ABS` is based on the KeY 2.0 platform—a verification system for Java. To be able to reuse most parts of the system, we had to generalize various


The Start button for automatic verification
KeY 0.1.0
No solver available
Proof Management
Proofs
Env. with model bank example@
Preserves Invariant_Acco
Proof
Goals
Proof Tree
1:OPEN GOAL
Current Goal
An ABSDL formula generated by keY-ABS
Integrated Deductive Software Design: Ready (Hint: type F3 to search in proof trees or sequents.)

```
==>
 \forall ABSAnyInterface caller;
   (   !caller = null
    -> \forall int x_0;
         {x:=x_0}
           {history:=seqConcat(history,
                                seqSingleton(invocREv(caller,
                                                      this,
                                                      future,
                                                      Account.Account::deposit#ABS.StdLib.Int,
                                                      seqConcat(seqEmpty, seqSingleton(x)))))}
             (      wellFormed(heap)
                & wfHist(history)
                & !this = null
                & Precondition_of_Account.Account::deposit#ABS.StdLib.Int(history, heap, this, x)
                & CInv(history, heap, this)
              ->  \[{   methodframe(source <- Account.Account::deposit#ABS.StdLib.Int, return <- (var: result, fut: future):{

                          this.balance = this.balance + x;
                         return
                         this.balance;
                       }
                    }\]CInv(history, heap, this)))
```

Figure 3.1: A screen shot of KeY-ABS.

subsystems and to abstract away from their Java specifics. For instance, the rule application logic of KeY made several assumptions which are valid for Java but not for other programming languages. Likewise, the specification framework of KeY, even though it provided general interfaces for contracts and invariants, made implicit assumptions that were insufficient for our communication histories and needed to be factored out. After refactoring, the KeY system provides core subsystems (rule engine, proof construction, search strategies, specification language, proof management etc.) that are independent of the specific program logic or target language. These are then extended and adapted by the ABS and Java backends.

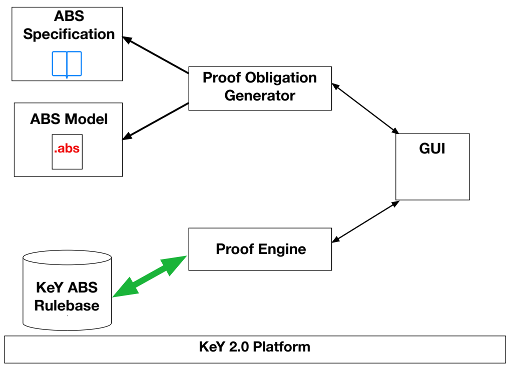


Figure 3.2: The architecture of KeY-ABS

The proof obligation generator needs to parse the source code of the ABS model and the specification. For the source code we use the parser as provided by the ABS toolkit [19] with no changes. The resulting abstract syntax tree is then converted into KeY's internal representation. The specification parser is an extension of the parser for ABSDL logic formulas and is an adapted version of the parser for JavaDL [1]. The rule base for ABSDL reuses the language-independent theories of the KeY tool, such as arithmetic, sequences and first-order logic. The rules for symbolic execution have been written from scratch for ABS as well as the formalisation of the history datatype.

3.4 The Usage of KeY-ABS

The compositionality of the reasoning system explained in Chapter 2 allows to scale verification to non-trivial programs, because it is possible to specify and verify each ABS method separately, without the need for a global invariant. KeY-ABS follows the Design-by-Contract paradigm with a focus on specification of class invariants for ABS programs.

A history-based class invariant in ABSDL can relate the state of an object to the local history of the system. A simple banking system is verified in [2] by KeY-ABS, where an invariant ensures that the value of the account balance (a class attribute) always coincides with the value returned by the most recent call to a deposit or withdraw method (captured in the history).

A history-based class invariant in ABSDL can also express temporal or structural properties of the history. To formalize this kind of class invariant, quantifiers and indices of sequences are used to locate the events at certain positions of the history. In Chapter 4 we show a case study using `KeY-ABS` to verify an ABS model of a Network-on-Chip (NoC) packet switching platform [13]. We proved that the NoC model drops no packets. Both styles of class invariants mentioned above were used. The `KeY-ABS` verification approach to the NoC case study deals with an *unbounded* number of objects and is valid for generic NoC models for any m $\times$ n mesh in the ASPIN chip as well as any number of sent packets.

Based on the theory of compositionality in Chapter 2, we prove *global* safety properties of the system using *local* rules and symbolic execution of `KeY-ABS`. In contrast to model checking this allows us to deal effectively with unbounded target systems without suffering from state explosion.

Chapter 4

The NoC Verification Case Study with KeY-ABS

4.1 Introduction

This chapter presents a case study on *scalable* formal verification of the behavior of distributed systems. We create a formal, executable model of a *Network-on-Chip* (NoC) packet switching platform [13] called ASPIN (Asynchronous Scalable Packet Switching Integrated Network) [17]. ASPIN has physically distributed routers in each core. Each router is connected to four other neighboring routers and each core is locally connected to one router. ASPIN routers are split into five separate modules (north, south, east, west, and local) with ports that have input and output channels and buffers. Each input channel is provided with an independent FIFO buffer. Packets arriving from different neighboring routers (and from the local core) are stored in the respective FIFO buffer. Communication between routers is established using a four-phase handshake protocol. The protocol uses request and acknowledgment messages between neighboring routers to transfer a packet. This is a practically relevant system whose correctness is of great importance for the network infrastructures where it is deployed.

The main contributions of this work are as follows: (i) a formal model of a generic $m \times n$ mesh ASPIN chip in ABS with unbounded number of packets, as well as a packet routing algorithm; (ii) the formal specification of a number of safety properties which together ensure that no packets are lost; (iii) formal proofs, done with `KeY-ABS`, that the ABS model of ASPIN fulfills these safety properties.[1]

4.2 The NoC Model in ABS

We model each router as an object that communicates with other routers through asynchronous method calls. The abstract data types used in our model are given in Fig. 4.1. We abstract away from the local communication to cores, so each router has four ports and each port has an input and output channel, the identifier `rId` of the neighbor router and a buffer. Packets are modeled as pairs that contain the packet identifier and the final destination coordinate.

The ABS model of a router is given in Fig. 4.2. The method `setPorts` initializes all the ports in a router and connects it with the corresponding neighbor routers. Packets are transferred using a protocol expressed in our model with two methods `redirectPk` and `getPk`. The internal method `redirectPk` is called when a router wants to redirect a packet to a neighbor router. The X-first routing algorithm (implemented by the function `xFirstRouting`) decides which port `direc` (and as a consequence which neighbor router) to choose. The parameter `srcPort` determines in which input buffer the packet is temporarily and locally stored. As part of the communication protocol, the input channel of `srcPort` and the output channel of `direc` are blocked until the neighbor router confirms that it has gotten the packet, using `f = r!getPk(...); await f?` statements

[1] The complete model with all formal specifications and proofs is available at `https://www.se.tu-darmstadt.de/se/group-members/crystal-chang-din/noc`.

```
type Pos = Pair<Int, Int>; // (x,y) coordinates
type Packet = Pair<Int, Pos>; // (id, destination)
type Buffer = Int;
data Direction = N | W | S | E | NONE ; // north, west, south, east, the direction for not moving
data Port = P(Bool inState , Bool outState, Router rId, Buffer buff);
        // (input port state, output port state, neighbor router id, buffer size)
type Ports = Map<Direction, Port>;
```

Figure 4.1: ADTs for the ASPIN model in `ABS`

to simulate request and acknowledgment messages (here `r` is the Id of the neighbor router). The method `getPk` checks if the final destination of the packet is the current router, if so, it stores the packet, otherwise it temporarily stores the packet in the `srcPort` buffer and redirects it. The model uses standard library functions for maps and sets (e.g, `put`, `lookup`, etc.) and observers as well as other functions over the ADTs (e.g., `addressPk`, `inState`, `decreaseBuff`, etc.).

4.3 Formal Specification and Verification of the Case Study

We formalize and verify global safety properties about our ABS NoC model in ABSDL using the `KeY-ABS` verification tool. This excludes any possibility of error at the level of the ABS model. According to Chapter 3 Section 3.4, we specify history-based *class invariants* which relate the state of an object to the local history of the system and express temporal or structural properties of the history. Our verification approach uses local reasoning about `RouterImp` objects and establishes a system invariant from the proof results.

4.3.1 Local Reasoning

We present the class invariants for `RouterImp` in Lemma 4.3.1 and 4.3.2 and we show the proof obligations verified by `KeY-ABS` that result from the reasoning of our model against the class invariants.

Lemma 4.3.1 *Whenever a router R terminates an execution of the `getPk` method, then R must either have sent an internal invocation to redirect the packet or have stored the packet in its `receivedPks` set.*

We formalize this lemma as an ABSDL formula (slightly beautified):

$$\begin{array}{l}
\forall i_1, u \,.\, 0 \leq i_1 < len(h) \wedge futEv(\texttt{this}, u, \texttt{getPk}, \_) = at(h, i_1) \\
\Rightarrow \\
\quad \exists i_2, pk \,.\, 0 \leq i_2 < i_1 \wedge invREv(\_, \texttt{this}, u, \texttt{getPk}, (pk, \_)) = at(h, i_2) \wedge \\
\quad ((dest(pk) \neq address(this) \Rightarrow \\
\qquad \exists i_3 \,.\, i_2 < i_3 < i_1 \wedge invEv(\texttt{this}, \texttt{this}, \_, \texttt{redirectPk}, (pk, \_)) = at(h, i_3)) \vee \\
\quad (dest(pk) = address(this) \Rightarrow pk \in \texttt{receivedPks}))
\end{array}$$

where "_" denotes a value that is of no interest. The function $len(s)$ returns the length of the sequence s, the function $at(s, i)$ returns the element located at the index i of the sequence s, the function $dest(pk)$ returns the destination address of the packet pk, and $address(r)$ returns the address of the router r.

This formula expresses that for every *future event* ev_1 of `getPk` with future identifier u found in history h we can find by pattern matching with u in the preceding history a corresponding *invocation reaction event* ev_2 that contains the sent packet pk. If `this` router is the destination of pk, then pk must be in its `receivedPks` set, otherwise an *invocation event* of `redirectPk` containing pk must be found in the history between events ev_1 and ev_2.

Lemma 4.3.2 *Whenever a router R terminates an execution of `redirectPk`, the input channel of `srcPort` and the output channel of `direc` are released.*

```
interface Router{
    Unit setPorts(Router e, Router w, Router n, Router s);
    Unit getPk(Packet pk, Direction srcPort);}

class RouterImp(Pos address, Int buffSize) implements Router {
   Ports ports = EmptyMap;
   Set<Packet> receivedPks = EmptySet; // received packages

   Unit setPorts(Router e, Router w, Router n, Router s){
     ports = map[Pair(N, P(True, True, n, 0)), Pair(S, P(True, True, s, 0)),
                 Pair(E, P(True, True, e, 0)), Pair(W, P(True, True, w, 0))];}

   Unit getPk(Packet pk, Direction srcPort){
     if (addressPk(pk) != address) {
       await buff(lookup(ports,srcPort)) < buffSize;
       ports = put(ports,srcPort,increaseBuff(lookup(ports,srcPort)));
       this!redirectPk(pk,srcPort);}
     else { // record that packet was successfully received
       receivedPks = insertElement(receivedPks, pk); } }

   Unit redirectPk(Packet pk, Direction srcPort){
     Direction direc = xFirstRouting(addressPk(pk), address);
     await (inState(lookup(ports,srcPort)) == True)
           && (outState(lookup(ports,direc)) == True);
     ports = put(ports, srcPort, inSet(lookup(ports, srcPort), False));
     ports = put(ports, direc, outSet(lookup(ports, direc), False));
     Router r = rId(lookup(ports, direc));
     Fut<Unit> f = r!getPk(pk, opposite(direc)); await f?;
     ports = put(ports, srcPort, decreaseBuff(lookup(ports, srcPort)));
     ports = put(ports, srcPort, inSet(lookup(ports, srcPort), True));
     ports = put(ports, direc, outSet(lookup(ports, direc), True));}}
```

Figure 4.2: A model of an ASPIN router using ABS

Again, we formalize this lemma as an ABSDL formula:

$$\begin{gathered}\forall u \,.\, \mathit{futEv}(\texttt{this}, u, \texttt{redirectPk}, \_) = \mathit{at}(h, \mathit{len}(h) - 1)\\ \Rightarrow\\ \exists i_1, i_2, pk, \texttt{srcP}, \texttt{dirP} \,.\, 0 < i_1 < i_2 < \mathit{len}(h) - 1 \wedge\\ (\mathit{invREv}(\texttt{this}, \texttt{this}, u, \texttt{redirectPk}, (pk, \texttt{srcP})) = \mathit{at}(h, i_1) \wedge\\ \mathit{invEv}(\texttt{this}, \_, \_, \texttt{getPk}, (pk, \mathit{opposite}(\texttt{dirP}))) = \mathit{at}(h, i_2)) \wedge\\ (\mathit{inState}(\mathit{lookup}(\mathit{ports}, \texttt{srcP})) \wedge \mathit{outState}(\mathit{lookup}(\mathit{ports}, \texttt{dirP})))\end{gathered}$$

This formula expresses that whenever the last event in the history h is a *future event* of `redirectPk` method, by pattern matching with the same future and packet in the previous history, we can find the corresponding *invocation reaction event* and the *invocation event*. In these two events we filter out the source port $srcP$ and the direction port $dirP$ used in the latest run of `redirectPk`. The input channel of `srcP` and the output channel of `dirP` must be released in the current state. This invariant captures the properties of the current state and is prefix-closed. With KeY-ABS we proved that the `RouterImp` class of our model satisfies this invariant.

The statistics of verifying these two invariants is given below. For each of the three methods of the `RouterImp` class we show it satisfies both invariants.

# nodes – # branches	**setPorts**	**getPk**	**redirectPk**
Lemma. 4.3.1	1638–12	11540–108	27077–200
Lemma. 4.3.2	214–1	1845–11	4634–34

`KeY-ABS` provides heuristics and proof strategies that automate large parts of proof construction. The remaining user input typically consists of universal and existential quantifier instantiations.

4.3.2 System Specification

According to the theory of compositional reasoning provided in Chapter 2, we establish a system invariant for the NoC model based on the invariants of the `RouterImp` class proved by `KeY-ABS`:

Theorem 4.3.3 *Whenever a router R releases a pair of input and output channels used for redirecting a receiving packet, the next router of R must either have sent an internal invocation to redirect the packet or have stored the packet in its `receivedPks` set. Hence, the network does not drop any packets.*

4.4 Conclusion

ABS has been developed with the explicit aim to permit scalable verification of detailed, precisely modeled, executable, concurrent systems. Our paper in Appendix. C shows that this claim is indeed justified. In addition it advances the state-of-the-art with the first successful verification of a generic NoC model that has an unbounded number of nodes and packets. This has been achieved with manageable effort and thus shows that deductive verification is a viable alternative to model checking for the verification of concurrent systems that can effectively deal with state explosion.

Chapter 5

Deadlock Analysis

5.1 Introduction

Modern systems are designed to support a high degree of parallelism by letting as many system components as possible operate concurrently. When such systems also exhibit a high degree of resource and data sharing then deadlocks represent an insidious and recurring threat. In particular, 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 with unbounded (mutual) recursion and dynamic resource creation. A paradigm case is an adaptive system that creates an unbounded number of processes such as server applications. In these systems, the interaction protocols are extremely complex and state-of-the-art solutions either give imprecise answers or do not scale.

In order to augment precision and scalability we propose a modular framework—which is described in the paper in Appendix D—that allows several techniques to be combined. We meet the scalability requirement by designing a front-end inference system that automatically extracts abstract behavioral descriptions pertinent to deadlock analysis, called *contracts*, from code. The inference system is *modular* because it (partially) supports separate inference of modules. To meet precision of contracts' analysis, as a proof-of-concept we define and implement two different techniques: (i) an evaluator that computes a fixpoint semantics and (ii) an evaluator using abstract model checking.

Our framework targets ABS. Because of the presence of explicit synchronisation operations, the analysis of deadlocks in ABS is more fine-grained than in thread-based languages (such as `Java`). However, as usual with (concurrent) programming languages, analyses are hard and time-consuming because most parts of the code are irrelevant for the properties one intends to derive. For this reason, we design an inference system that *automatically extracts contracts* from ABS code. These contracts are similar to those ranging from languages for session types [10] to process contracts [14] and to calculi of processes as Milner's CCS or pi-calculus [15, 16]. The inference system mostly collects method behaviors and uses constraints to enforce consistencies among behaviors. Then a standard semiunification technique is used for solving the set of generated constraints.

Since our inference system addresses a language with asynchronous method invocations, it is possible that a method triggers behaviors that will last *after* the execution of the method has been completed (and therefore will contribute to *future* deadlocks). In order to support a more precise analysis, we split contracts of methods in *synchronised* and *unsynchronised contracts*, with the intended meaning that the former collect the invocations that are explicitly synchronised in the method body (namely there is a blocking operation that waits for the result of the asynchronous invocation) and the latter ones collect the other invocations (those that are not synchronised within the caller's method and keep executing even after the caller's method execution terminates).

Our contracts feature recursion and resource creation; therefore their underlying models are infinite state and their analysis cannot be exhaustive. We propose two techniques for analysing contracts (and to show the modularity of our framework). The first one, is a *fixpoint technique* on models with a limited capacity of name creation (possibly resorting to a saturation technique since contract pairs models may be infinite-state). This entails fixpoint existence and finiteness of models. While we lose precision, our technique is sound (in some cases, this technique may signal false positives). The second technique is an *abstract model checking* that consists of computing contract models by expanding their invocations. Instead of a saturation technique, which introduces inaccuracies, we exploit a generalisation of permutation theory that let us decide when to stop the evaluation with the guarantee that if no circular dependency has been found up to that moment then it will not appear afterwards. That stage corresponds to the order of an associated permutation. It turns out that this technique is suited for so-called linear recursive contract class tables. When the recursions are linear, this technique is precise—namely the contract contains a deadlock if and only if the technique finds a circular dependency—, while it is over-approximating in general. Notice that the precision of the analysis is not evaluated with respect to the program, but with respect to the contract.

5.2 The `DF4ABS` tool

We extended the ABS suite [19] with an implementation of our deadlock analysis framework (at the time of writing the suite has only the fixpoint analyser, the full framework is available at `http://df4abs.nws.cs.unibo.it`). The `DF4ABS` tool is built upon the abstract syntax tree (AST) of the ABS type checker, which allows us to exploit the type information stored in every node of the tree. This simplifies the implementation of several contract inference rules. There are four main modules that comprise `DF4ABS`:

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 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 defined in the paper.

2. *Constraint Solving* is done by a generic semi-unification solver implemented in Java, following the algorithm defined in [12]. When the solver terminates (and no error is found), it produces a substitution that satisfies the input constraints. Applying this substitution to the generated contracts produces the abstract class table and the contract of the main function of the program.

3. *Fixpoint Analysis* uses dynamic structures to store the lam (the abstract models of programs in term of their synchronisation dependencies) of every method contract (because lams become larger and larger as the analysis progresses). At each iteration of the analysis: i) a number of fresh cog names is created, ii) the states are updated according to what is prescribed by the contract, and iii) the tool checks whether a fixpoint has been reached. Saturation starts when the number of iterations reaches a maximum value (that may be customised by the user). In this case, since the precision of the algorithm degrades, the tool signals that the answer may be imprecise. To detect whether a relation in the fixpoint lam contains a circular dependency, we run Tarjan's algorithm [18] for connected components of graphs and we stop the algorithm when a circularity is found.

4. *Abstract model checking* algorithm for deciding the circularity-freedom problem in linear recursive contract class tables performs the following steps. (*i*) *Find (linear) recursive methods*: by parsing the contract class table we create a graph where nodes are function names and, for every invocation of `D.n` in the body of `C.m`, there is an edge from `C.m` to `D.n`. Then a standard depth first search associates to every node a path of (mutual) recursive invocations (the paths starting and ending at that node, if any). The contract class table is linear recursive if every node has at most one associated path. (*ii*) *Computation of the orders*: given the list of recursive methods, we compute the corresponding orders. (*iii*) *Evaluation process*: the contract pair corresponding to the main function is evaluated till every

program	lines	DF4ABS/fixpoint result	time	DF4ABS/model-check result	time	DECO result	time
PingPong	61	✓	0.311	✓	0.046	✓	1.30
MultiPingPong	88	D	0.209	D	0.109	D	1.43
BoundedBuffer	103	✓	0.126	✓	0.353	✓	1.26
PeerToPeer	185	✓	0.320	✓	6.070	✓	1.63
FAS Module	2645	✓	31.88	✓	39.78	✓	4.38

Table 5.1: Assessments of DF4ABS.

recursive function invocation has been unfolded up-to twice the corresponding order. (*iv*) *Detection of circularities*: this is performed with the same algorithm of the fixpoint analysis.

Regarding the computational complexity, the contract inference system runs in polynomial time with respect to the length of the program in most of the cases [12]. The fixpoint analysis is exponential in the number of cog names in a contract class table (because lams may double their size at every iteration). However, this exponential effect actually bites in practice. The abstract model checking is linear with respect to the length of the program as far as steps (i) and (ii) are concerned. Step (iv) is linear with respect to the size of the final lam. The critical step is (iii), which may be exponential with respect to the length of the program. Below, there is an overestimation of the computational complexity. Let

$\mathfrak{o}_{max}$ be the largest order of a recursive method contract (without loss of generality, we assume there is no mutual recursion).

m_{max} be the maximal number of function invocations in a body or in the contract of the main function.

An upper bound to the length of the evaluation till the saturated state is $\sum_{0\leq i\leq \ell}(2 \times \mathfrak{o}_{max} \times m_{max})^i$, where ℓ is the number of methods in the program. Let k_{max} be the maximal number of dependency pairs in a body. Then the size of the saturated state is $O(k_{max} \times (\mathfrak{o}_{max} \times m_{max})^{\ell})$, which is also the computational complexity of the abstract model checking.

5.3 Assessments

We tested DF4ABS on a number of medium-sized programs written for benchmarking purposes by ABS programmers and on an industrial case study based on the Fredhopper Access Server (FAS) [8]. The Table 5.1 reports the experiments: for every program we display the number of lines, whether the analysis has reported a deadlock (D) or not (✓), the time in seconds required for the analysis. Concerning time, we only report the time of the analysis of DF4ABS (and not the one taken by the inference) when they run on a QuadCore 2.4GHz and Gentoo (Kernel 3.4.9).

The rightmost column of the table in Figure 5.1 reports the results of another tool that has also been developed for the deadlock analysis of ABS programs: DECO [9]. This technique integrates a points-to analysis with an analysis returning (an over-approximation of) program points that may be running in parallel. As highlighted by the above table, the three tools return identical results regarding deadlock analysis, but are different with respect to performance. In particular the fixpoint and model-checking analysis of DF4ABS are comparable on small/mid-size programs, DECO appears less performant (except for PeerToPeer, where our model-checking analysis is quite slow because of the number of dependencies produced by the underlying algorithm). On the FAS module, our two analysis are again comparable, while DECO has a better performance (DECO's worst case complexity is cubic in the size of the input).

A few remarks about the precision of the techniques follow. DF4ABS/model-check is the most powerful tool we are aware of for linear recursive contract class tables. We found and reported in the paper examples for which it correctly detects the deadlock-freedom, while DF4ABS/fixpoint and DECO signal a false positive.

However, DF4ABS/model-check is not defined on non-linear recursive contract class tables. In these cases, DF4ABS/model-check fails to analyse while DF4ABS/fixpoint and DECO successfully recognise as deadlock-free. Nonetheless, studies show that program are generally linearly recursive, so that restriction is not too strong in practice. This substantiates the usefulness of our technique in these programs; the analysis of a wider range of programs is matter of future work.

5.4 Conclusive remarks

DF4ABS, being modular, may be integrated with other analysis techniques. In fact, we have recently defined an even more powerful technique (which is decidable also on non-linear recursive lams), described in Deliverable D2.1 [5]. This technique was a later improvement of our approach, thus was not described in the paper attached in Appendix D, nor in the version of the tool integrated with the ABS suite. But the theory has been completely investigated [11] and its implementation is part of the new release of the tool available at `http://df4abs.nws.cs.unibo.it`.

Bibliography

[1] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software. The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[2] Richard Bubel, Antonio Flores Montoya, and Reiner Hähnle. Analysis of executable software models. In Marco Bernardo, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen, and Ina Schaefer, editors, *Executable Software Models: 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinoro, Italy*, volume 8483 of *Lecture Notes in Computer Science*, pages 1–27. Springer-Verlag, June 2014.

[3] Dave Clarke, Nikolay Diakov, Reiner Hähnle, Einar Broch Johnsen, Ina Schaefer, Jan Schäfer, Rudolf Schlatte, and PeterY.H. Wong. Modeling spatial and temporal variability with the hats abstract behavioral modeling language. In Marco Bernardo and Valérie Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 417–457. Springer Berlin Heidelberg, 2011.

[4] Frank de Boer, Dave Clarke, and Einar Johnsen. A complete guide to the future. In *Progr. Lang. and Systems*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer-Verlag, 2007.

[5] Behavioural Interfaces for Virtualized Services, September 2014. Deliverable D2.1 of project FP7-610582 (ENVISAGE), available at `http://www.envisage-project.eu`.

[6] Crystal Chang Din and Olaf Owe. Compositional reasoning about active objects with shared futures. *Formal Aspects of Computing*, pages 1–22, 2014.

[7] Johan Dovland, Einar Broch Johnsen, and Olaf Owe. Verification of concurrent objects with asynchronous method calls. In *Proceedings of the IEEE International Conference on Software Science, Technology & Engineering (SwSTE'05)*, pages 141–150. IEEE Computer Society Press, February 2005.

[8] Requirement elicitation, August 2009. Deliverable 5.1 of project FP7-231620 (HATS), available at `http://www.hats-project.eu/sites/default/files/Deliverable51_rev2.pdf`.

[9] Antonio Flores-Montoya, Elvira Albert, and Samir Genaim. May-happen-in-parallel based deadlock analysis for concurrent objects. In *Proc. FORTE/FMOODS 2013*, volume 7892 of *Lecture Notes in Computer Science*, pages 273–288. Springer-Verlag, 2013.

[10] Simon Gay and Malcolm Hole. Subtyping for session types in the π-calculus. *Acta Informatica*, 42(2-3):191–225, 2005.

[11] Elena Giachino, Naoki Kobayashi, and Cosimo Laneve. Deadlock detection of unbounded process networks. In *Proceedings of CONCUR 2014*, volume 8704 of *Lecture Notes in Computer Science*, pages 63–77. Springer-Verlag, 2014.

[12] Fritz Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, April 1993.

[13] Shashi Kumar, Axel Jantsch, Mikael Millberg, Johnny Öberg, Juha-Pekka Soininen, Martti Forsell, Kari Tiensyrjä, and Ahmed Hemani. A network on chip architecture and design methodology. In *2002 IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2002), 25-26 April 2002, Pittsburgh, PA, USA*, pages 117–124, 2002.

[14] Cosimo Laneve and Luca Padovani. The *must* preorder revisited. In *Proc. CONCUR 2007*, volume 4703 of *Lecture Notes in Computer Science*, pages 212–225. Springer-Verlag, 2007.

[15] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[16] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Inf. and Comput.*, 100:1–77, 1992.

[17] Abbas Sheibanyrad, Alain Greiner, and Ivan Miro Panades. Multisynchronous and fully asynchronous NoCs for GALS architectures. *IEEE Design & Test of Computers*, 25(6):572–580, 2008.

[18] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[19] Peter Y. H. Wong, Elvira Albert, Radu Muschevici, José Proença, Jan Schäfer, and Rudolf 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.

Glossary

ABS Abstract behavioral Specification language. An executable class-based, concurrent, object-oriented modelling language based on Creol, created for the HATS project.

Behavioral Interface The intended behavior of programs such as functional behavior and resource consumption can be expressed in the behavioral interface. Formal specifications of program behavior is useful for precise documentation, for the generation of test cases and test oracles, for debugging, and for formal program verification.

Communication History The communication history h of a system of objects S is a sequence of events, such that each event in h is generated by an object in S.

Contract Abstract specification of a program's behavior at runtime, used to perform specific analysis on the program, like deadlock detection or resource consumption analysis.

Contract Class Table Given an ABS program, its contract class table associates a contract to each method of each class in the program.

Linear Recursive Contract Class Table A contract class table where (mutual) recursive invocations in bodies of methods have at most one recursive invocation.

Observable Behavior The observable behavior of an object is the interaction between the object and its environment which can be captured in the communication history over observable events.

Deductive Verification A process of reasoning from one or more statements (premises) to reach a logically certain conclusion. A proof may be seen as a tree with axioms as leaves and the main theorem as the root. Each internal node of the proof tree is a consequence of its immediate descendant nodes according to given proof rules.

Lam Main data structure of the Deadlock Analysis which stores all the sets of await and get synchronisations between cogs possibly generated by a method's execution.

Appendix A

A Sound Compositional Reasoning System in Dynamic Logic

DOI 10.1007/s00165-014-0322-y
BCS © 2014
Formal Aspects of Computing

Formal Aspects
of Computing

Compositional reasoning about active objects with shared futures

Crystal Chang Din[1] and Olaf Owe[2]

[1] Department of Computer Science, Technische Universität Darmstadt, Darmstadt, Germany
[2] Department of Informatics, University of Oslo, Oslo, Norway

Abstract. Distributed and concurrent object-oriented systems are difficult to analyze due to the complexity of their concurrency, communication, and synchronization mechanisms. The *future mechanism* extends the traditional method call communication model by facilitating sharing of references to futures. By assigning method call result values to futures, third party objects may pick up these values. This may reduce the time spent waiting for replies in a distributed environment. However, futures add a level of complexity to program analysis, as the program semantics becomes more involved. This paper presents a model for asynchronously communicating objects, where return values from method calls are handled by futures. The model facilitates invariant specifications over the locally visible communication history of each object. Compositional reasoning is supported and proved sound, as each object may be specified and verified independently of its environment. A kernel object-oriented language with futures inspired by the ABS modeling language is considered. A compositional proof system for this language is presented, formulated within dynamic logic.

Keywords: Distributed systems, Object orientation, Concurrent objects, Asynchronous communication, Shared futures, Operational semantics, Communication history, Compositional reasoning, Dynamic logic

1. Introduction

Distributed systems play an essential role in society today. However, quality assurance of distributed systems is non-trivial since they depend on unpredictable factors, such as different processing speeds of independent components. Therefore, it is highly challenging to test such distributed systems after deployment under different relevant conditions. These challenges motivates frameworks combining precise modeling and analysis with suitable tool support. In particular, *compositional verification systems* allow the different components to be analyzed independently from their surrounding components.

Object orientation is the leading framework for concurrent and distributed systems, recommended by the RM-ODP [Int95]. However, method-based communication between concurrent units may cause busy-waiting, as in the case of remote and synchronous method invocation, e.g., Java RMI [AY07]. Concurrent objects communicating by *asynchronous method calls* have been proposed as a promising framework to combine object-orientation and

Correspondence and offprint requests to: C. C. Din and O. Owe, E-mail: crystald@cs.tu-darmstadt.de and olaf@ifi.uio.no

Published online: 19 November 2014

distribution in a natural manner. Each concurrent object encapsulates its own state and processor, and internal interference is avoided as at most one process is executing on an object at a time. Asynchronous method calls allow the caller to continue with its own activity without blocking while waiting for the reply, and a method call leads to a new process on the called object. The notion of *futures* [BJH77, HJ85, LS88, YBS86] improves this setting by providing a decoupling of the process invoking a method and the process reading the returned value. By sharing *future identities*, the caller enables other objects to wait for method results. However, futures complicate program analysis since programs become more involved compared to semantics with traditional method calls, and in particular local reasoning is a challenge.

The execution of a distributed system can be represented by its *communication history* or *trace*; i.e., the sequence of observable communication events between system components [BS01, Hoa85]. At any point in time the communication history abstractly captures the system state [Dah87, Dah92]. In fact, traces are used in semantics for full abstraction results (e.g., [ÁGGS09, JR05]). The *local history* of an object reflects the communication visible to that object, i.e., between the object and its surroundings. A system may be specified by the finite initial segments of its communication histories, and a *history invariant* is a predicate which holds for all finite sequences in the set of possible histories, expressing safety properties [AS85].

In this work we consider a kernel object-oriented language, where futures are used to manage return values of method calls. Objects are concurrent and communicate asynchronously. We formalize object communication by a four event operational semantics, capturing shared futures, where each event is visible to only one object. Consequently, the local histories of two different objects share no common events, and history invariants can be established independently for each object. We present a *dynamic logic* proof system for class verification, facilitating independent reasoning about each class. A verified *class invariant* can be instantiated to each object of that class, resulting in an invariant over the local history of the object. Modularity is achieved as the independently derived history invariants can be composed to form *global* system specifications. Global history consistency is captured by a notion of history *well-formedness*. The formalization of object communication extends previous work [DDJO12] which considered concurrent objects and asynchronous communication, but without futures.

Paper overview. Section 2 presents a core language with shared futures. The communication model is presented in Sect. 3, and Sect. 4 defines the operational semantics. Sect. 5 presents the compositional reasoning system, and Sect. 6 contains related work and concludes the paper.

2. A core language with shared futures

A *future* is a placeholder for the return value of a method call. Each future has a unique identity which is *generated* when a method is invoked. The future is *resolved* upon method termination, by placing the return value of the method in the future. Thus, unlike the traditional method call mechanism, the callee does not send the return value directly back to the caller. However, the caller may keep a *reference* to the future, allowing the caller to *fetch* the future value once resolved. References to futures may be shared between objects, e.g., by passing them as parameters. After achieving a future reference, this means that third party objects may fetch the future value. Thus, the future value may be fetched several times, possibly by different objects. In this manner, shared futures provide an efficient way to distribute method call results to a number of objects.

For the purposes of this paper, we consider a core object-oriented language with futures, presented in Fig 1. It includes basic statements for first class futures, inspired by *ABS* [HAT]. Class instances are concurrent, encapsulating their own state and processor. Each method invoked on the object leads to a new process, and at most one process is executing on an object at a time. Object communication is *asynchronous*, as there is no explicit transfer of control between the caller and the callee. Methods are organized in classes in a standard manner. A class C takes a list of formal parameters $\overline{cp}$, defines fields $\overline{w}$, initialization block s and methods $\overline{M}$. There is read-only access to the parameters $\overline{cp}$. A method definition has the form $m(\overline{x})\{\mathbf{var}\ \ \overline{y};\ s;\ \mathbf{return}\ e\}$, ignoring type information, where $\overline{x}$ is the list of parameters, $\overline{y}$ an optional list of *method-local variables*, s is a sequence of statements, and the value of e is returned upon termination.

A future variable *fr* is declared by $\mathsf{Fut}<T>$ *fr*, indicating that *fr* may refer to futures which will eventually contain values of type T. The call statement *fr* $:= x!m(\overline{e})$ invokes the method m on object x with input values $\overline{e}$. The identity of the generated future is assigned to *fr*, and the calling process continues execution without waiting for *fr* to become resolved. The query statement $v :=$ *fr*? is used to fetch the value of a future. The statement blocks until *fr* is resolved, and then assigns the value contained in *fr* to v. The language contains additional statements for assignment, **skip**, conditionals, and sequential composition.

$$\begin{array}{lll}
Cl & ::= \textbf{class}\ C([T\ cp]^*)\ \{[T\ w\ [:= e]^?]^*\ s\ M^*\} & \text{class definition}\\
M & ::= T\ m([T\ x]^*)\ \{[\textbf{var}\ [T\ x]^*]^?\ s\ ;\ \textbf{return}\ e\} & \text{method definition}\\
T & ::= C \mid \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{String} \mid \mathsf{Void} \mid \mathsf{Fut}{<}T{>} & \text{types}\\
v & ::= x \mid w & \text{variables (local or field)}\\
e & ::= \textbf{null} \mid \mathsf{this} \mid v \mid cp \mid f(\overline{e}) & \text{pure expressions}\\
s & ::= v := e \mid fr := v!m(\overline{e}) \mid v := e? & \text{statements}\\
 & \quad \mid \textbf{skip} \mid \textbf{if}\ e\ \textbf{then}\ s\ [\textbf{else}\ s]^?\ \textbf{fi} \mid s; s & \\
 & \quad \mid \textbf{while}\ e\ \textbf{do}\ s\ \textbf{od} \mid v := \textbf{new}\ C(\overline{e}) &
\end{array}$$

Figure 1. Core language syntax, with C class name, cp formal class parameter, m method name, w fields, x method parameter or local variable, and where fr is a future variable. We let $[\ ]^*$ and $[\ ]^?$ denote repeated and optional parts, respectively, and $\overline{e}$ is a (possibly empty) expression list. Expressions e and functions f are side-effect free

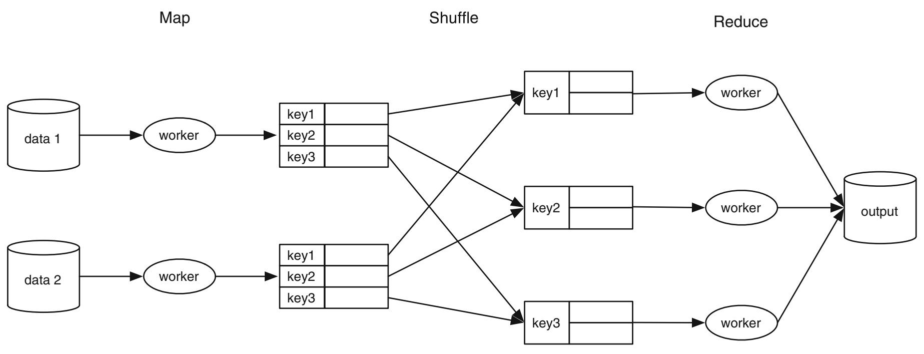


Figure 2. MapReduce model

We assume that call and query statements are well-typed. If x refers to an object where m is defined with no input values and return type Int, the following is a well-typed blocking method call: $\mathsf{Fut} < \mathsf{Int} > fr;\ \ \mathsf{Int}\ v;\ fr := x!m();\ \ v := fr?$.

To avoid blocking, *ABS* provides statements for process control, including a statement **await** $fr?$, which releases the current process as long as fr is not yet resolved. This gives rise to more efficient computing with futures. It is possible to add a treatment of process release statements as a straight forward extension of the present work, following the approach of [DDJO12]. We here focus on a core language for futures, with a simple semantics, avoiding specialized features such as process control. The core language ignores *ABS* features that are orthogonal to shared futures, including interface encapsulation, inheritance, local synchronous calls, and internal scheduling of processes by means of cooperative multitasking. We refer to the report version of this paper for a treatment of these issues [DDO12a].

2.1. The MapReduce example

In order to illustrate the usage of futures, we consider the problem of counting the number of occurrences of each word in a large collection of documents. We consider the computing model MapReduce in Fig. 2. MapReduce is invented and used heavily by Google for efficient distributed computing over large data sets [DG08]. It has three major steps: Map, Shuffle and Reduce. The Map phase runs over input data, which might be a database or some files, and output key-value pairs. The input data is split in parts so they can be processed by workers in parallel. The second step is the Shuffle phase, which collates values with the same key together. At last, the Reduce function is called by workers in parallel on the shuffled data distinguished by keys.

```
class Worker () implements WorkerI {

   List<Pair<String, Int>>
      invokeMap(String filename, List<String> content) {...}

   Int invokeReduce(String key, List<Int> value) {...}
   ...
}

class WorkerPool() implements WorkerPoolI {
   WorkerI getWorker() {// provides idle workers,
                        // or generates new workers if needed.}
}
```

Listing 1. Sketch of the classes Worker and WorkerPool

```
class MapReduce(WorkerPoolI wp) implements MapReduceI {

 List<Pair<String, Int>> mapReduce(
  List<Pair<String, List<String>>> files) {

  Set<Fut<List<Pair<String, Int>>>> fMapResults := EmptySet;
  Set<Pair<String, Fut<Int>>> fReduceResults := EmptySet;
  List<Pair<String, Int>> result := Nil;

  // Map phase //
  while(~isEmpty(files)) do
     ...
     WorkerI w := wp.getWorker();
     ...
     Fut<List<Pair<String, Int>>>
        fMap := w!invokeMap(filename, content);
     fMapResults := insertElement(fMapResults, fMap)
  od;

  // Shuffle phase //
  while(~emptySet(fMapResults)) do
     Fut<List<Pair<String, Int>>>
        fMapResult := take(fMapResults);
     ...
     List<Pair<String, Int>> mapResult := fMapResult?;
     ... // collates values with the same key together
  od;

  // Reduce phase //
  while(~emptySet(keys)) do
     ...
     WorkerI w := wp.getWorker();
     Fut<Int> fReduce := w!invokeReduce(key, values);
     fReduceResults := insertElement(
        fReduceResults, Pair(key, fReduce)) od;
  while(~emptySet(fReduceResults)) do
     Pair<String, Fut<Int>> reduceResult := take(fReduceResults);
     ...
     String key := fst(reduceResult);
     Fut<Int> fValue := snd(reduceResult);
     Int value := fValue?;
     result := Cons(Pair(key, value), result) od;
  return result;
 }
}
```

Listing 2. **The MapReduce class.** Here the notation $x := o.m(\overline{e})$ abbreviates $u := o!m(\overline{e});\ x := u?$ (for some fresh future u) to de-emphasize trivial usage of futures

We assume two interfaces, *WorkerI* and *MapReduceI*. The interface *WorkerI* is implemented by a class *Worker* shown in Listing 1, in which the method *invokeMap* takes a file and emits a list of pairs such that each word in the file is associated with a counting number: '1' in this example. For instance, if the content of the file is 'I am fine', the output of *invokeMap* is '(I,1),(am,1),(fine,1)'. The method *invokeReduce* in class *Worker* sums together all counts emitted for a particular word. For instance, if the word "am" appears twice, *invokeReduce* takes '(am, (1,1))' and outputs 2.

The interface *MapReduceI* is implemented by class *MapReduce*, shown in Listing 2. We here assume generic data types for sets, lists, and pairs, the latter with *fst* and *snd* to extract the first and second element, respectively.

The input to the method *mapReduce* is a list of files each starts with a filename and contains a list of words, i.e. the content of the file. Each file are handled by a worker in parallel. To achieve concurrency, for each file the object of *MapReduce* calls asynchronously the method *invokeMap* on the assigned worker *w*. This is realized by the statement *fMap := w!invokeMap(filename, content)*. The function *insertElement* collects all the futures into a set *fMapResults*. Next is the Shuffle phase. The function *take* randomly extracts an element from a set. The method *mapReduce* waits upon each future, gets the results from each future: *mapResult := fMapResult?*, and collates all the values with the same key, i.e. word, together. For instance, '(I,1),(am,1),(who,1),(I,1),(am,1)' is shuffled to '(I,(1,1)),(am,(1,1)),(who,(1))'. In the Reduce phase, each 'key' and the corresponding values are handled by a worker in parallel. In the same way as the Map phase for achieving concurrency, the first part of the reduce phase calls asynchronously the method *invokeReduce* on the assigned worker *w*. This is realized by the statement *fReduce := w!invokeReduce(key, values)*. The function *insertElement* collects all the futures into a set *fReduceResults*. At the very last, the method *mapReduce* waits upon each future, gets the results from each future: *value := fValue?*, and return the number of occurrences of each word in a large collection of files.

Here the future mechanism is exploited to make an efficient implementation, avoiding blocking calls on the workers: The Map phase is not waiting for the workers to do *invokeMap*, and is storing future identities only, thereby allowing many workers to start and work concurrently. Likewise in the loop calling *invokeReduce*, only futures identities are recorded. Blocking is delayed to phases where the future value information is actually needed.

2.1.1. The intentional reasoning about the MapReduce example

The implementation of MapReduce must guarantee the accuracy of the output. Namely, the summation of the occurrences of each word in the collection of documents is correct. However, it is not straight forward to verify this system property. The steps of calculation take place in different components in parallel: the Worker objects execute either the Map phase or the Reduce phase, and the MapReduce object shuffles the data and collects the result from the Worker objects. If we only prove the functional correctness of each class, it is not strong enough to prove this system property. Compositional reasoning is therefore required. We need a formalism to capture the interaction (order) between the components such that we are able to derive the system property from the local reasoning of each components. In the end of this paper, we will present a compositional proof of MapReduce which does provide correct number of occurrences of each word in the collection of documents.

3. Observable behavior

In this section we describe a communication model for concurrent objects communicating by means of asynchronous message passing and futures. The model is defined in terms of the *observable* communication between objects in the system. We consider how the execution of an object may be described by different *communication events* which reflect the observable interaction between the object and its environment. The observable behavior of a system is described by communication histories over observable events [BS01, Hoa85].

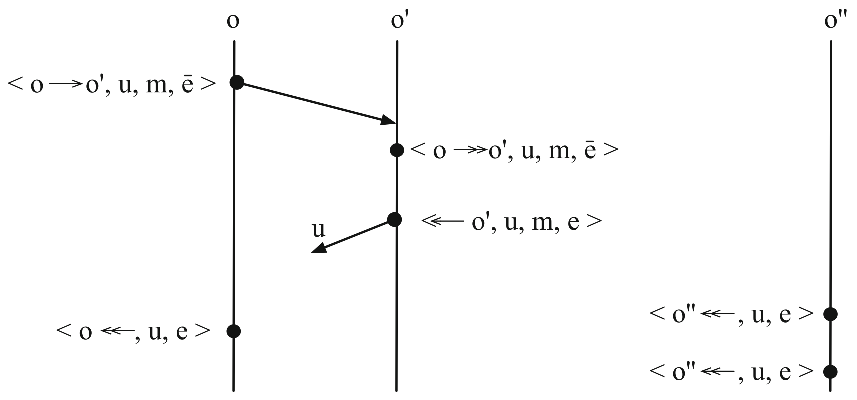


Figure 3. A method call cycle: object o calls a method m on object o' with *future* u. The events on the *left-hand side* are visible to o, those in the *middle* are visible to o', and the ones on the *right-hand side* are visible to o''. There is an arbitrary delay between message receiving and reaction

3.1. Communication events

Since message passing is asynchronous, we consider separate events for method invocation, reacting upon a method call, resolving a future, and for fetching the value of a future. Each event is observable to only one object, which is the one that *generates* the event. The events generated by a method call cycle is depicted in Fig. 3. The object o calls a method m on object o' with input values $\overline{e}$ and where u denotes the future identity. An invocation message is sent from o to o' when the method is invoked. This is reflected by the *invocation event* $\langle o \rightarrow o', u, m, \overline{e}\rangle$ generated by o. An *invocation reaction event* $\langle o \twoheadrightarrow o', u, m, \overline{e}\rangle$ is generated by o' once the method starts execution. When the method terminates, the object o' generates the *future event* $\langle \leftarrow o', u, m, e\rangle$. This event reflects that u is resolved with return value e. The *fetching event* $\langle o \twoheadleftarrow, u, e\rangle$ is generated by o when o fetches the value of the resolved future. Since future identities may be passed to other objects, e.g, o'', this object may also fetch the future value, reflected by the event $\langle o'' \twoheadleftarrow, u, e\rangle$, generated by o''. The creation of an object o' by an object o is reflected by the event $\langle o \xrightarrow{new} o', C, \overline{e}\rangle$, where o' is the instance of class C and $\overline{e}$ are the actual values for the class parameters. Let type Mid include all method names, and let Data be the supertype of all values occurring as actual parameters, including future identities Fid and object identities Oid.

Definition 3.1 (*Events*) Let *caller*, *callee*, *receiver* : Oid, *future* : Fid, *method* : Mid, *class* : Cls, *args* : List[Data], and *result* : Data. Communication events Ev include:

- *Invocation events* $\langle caller \rightarrow callee, future, method, args\rangle$, generated by *caller*.
- *Invocation reaction events* $\langle caller \twoheadrightarrow callee, future, method, args\rangle$, generated by *callee*.
- *Future events* $\langle \leftarrow callee, future, method, result\rangle$, generated by *callee*.
- *Fetching events* $\langle receiver \twoheadleftarrow, future, result\rangle$, generated by *receiver*.
- *Creation events* $\langle caller \xrightarrow{new} callee, class, args\rangle$, generated by *caller*.

Events may be decomposed by functions. For instance, _.*result* : Ev → Data is well-defined for future and fetching events, e.g., $\langle \leftarrow o', u, m, e\rangle.result = e$.

For a method invocation with future u, the ordering of events depicted in Fig. 3 is described by the following regular expression (using · for sequential composition of events)

$$\langle o \rightarrow o', u, m, \overline{e}\rangle \cdot \langle o \twoheadrightarrow o', u, m, \overline{e}\rangle \cdot \langle \leftarrow o', u, m, e\rangle[\cdot\langle \_ \twoheadleftarrow, u, e\rangle]^*$$

for some fixed o, o', m, $\overline{e}$, e, and where _ denotes an arbitrary value. This implies that the result value may be read several times, each time with the same value, namely that given in the preceding future event.

3.2. Communication histories

The execution of a system up to present time may be described by its history of observable events, defined as a sequence. A sequence over some type T is constructed by the empty sequence ε and the right append function $\_ \cdot \_ : \mathsf{Seq}[T] \times T \rightarrow \mathsf{Seq}[T]$ (where "_" indicates an argument position). The choice of constructors gives rise to generate inductive function definitions, in the style of [Dah92]. Projection, $\_/\_ : \mathsf{Seq}[T] \times \mathsf{Set}[T] \rightarrow \mathsf{Seq}[T]$ is defined inductively by $\varepsilon/s \triangleq \varepsilon$ and $(a \cdot x)/s \triangleq \texttt{if } x \in s \texttt{ then } (a/s) \cdot x \texttt{ else } a/s \texttt{ fi}$, for $a : \mathsf{Seq}[T]$, $x : T$, and $s : \mathsf{Set}[T]$, restricting a to the elements in s. We use dot notation to extract components from record-like structures, for instance $\langle o \rightarrow o', f, m, \overline{e} \rangle.\mathsf{callee}$ is o', and also lift the dot notation to sequences. For a sequence h of events, $h/ \leftarrow$ is the subsequence of invocation events, and $(h/ \leftarrow).callee$ is the sequence of callee elements from these invocation events.

A *communication history* for a set S of objects is defined as a sequence of events generated by the objects in S. We say that a history is *global* if S includes all objects in the system.

Definition 3.2 (*Communication histories*) The communication history h of a system of objects S is a sequence of type $\mathsf{Seq}[\mathsf{Ev}]$, such that each event in h is generated by an object in S.

We observe that the *local history* of a single object o is achieved by restricting S to the single object, i.e., the history contains only elements generated by o. For a history h, we let h/o abbreviate the projection of h to the events generated by o. Since each event is generated by only one object, it follows that the local histories of two different objects are disjoint.

Definition 3.3 (*Local histories*) For a global history h and an object o, the projection h/o is the local history of o.

4. Operational semantics

Rewriting logic [Mes92] is a logical and semantic framework in which concurrent and distributed systems can be specified in an object-oriented style. Unbounded data structures and user-defined data types are defined in this framework by means of equational specifications. Rewriting logic extends membership equational logic with *rewrite rules*, so that in a *rewrite theory*, the *dynamic* behavior of a system is specified as a set of rules on top of its *static* part, defined by a set of equations. Informally, a *labeled conditional rewrite rule* is a transition $l : \ t \longrightarrow t' \ \textbf{if} \ cond$, where l is a *label*, t and t' are terms over typed variables and function symbols of given arities, and *cond* is a condition that must hold for the transition to take place. Rewrite rules are used to specify local transitions in a system, from a state fragment that matches the pattern t, to another state fragment that is an instance of the pattern t'. Rules are selected nondeterministic if there are at least two rule instantiations with left-hand sides matching overlapping fragments of a term. Concurrent rewriting is possible if the fragments are non-overlapping. Furthermore, matching is made modulo the properties of the function symbols that appear in the rewrite rule, like associativity, commutativity, identity (ACI), which introduces further nondeterminism. The Maude tools [CDE+07] allow simulation, state exploration, reachability analysis, and LTL model checking of rewriting logic specifications. The state of a concurrent object system is captured by a *configuration*, which is an ACI multiset of *units* such as objects and messages, and other relevant system parts, which in our case includes futures. Concurrency is then supported in the framework by allowing concurrent application of rules when there are non-overlapping matches of left-hand sides. The following context rule, which is implicit in rewriting logic, describes interleaving semantics (letting G, G_1, G_2 denote subconfigurations):

$$context\ rule \quad \frac{G_1 \rightarrow G_2}{G\ G_1 \longrightarrow G\ G_2}$$

4.1. Operational rules

For our purpose, a *configuration* is a multiset of (concurrent) objects, classes, messages, futures, as well as a representation of the global history. We use blank-space as the multiset constructor, allowing ACI pattern matching. Objects have the form $\textbf{object}(Id : o, A)$ where o is the unique identity of the object and A is a set of semantic attributes, including

$Cl : c$	the class c of the object,
$Pr : s$	the remaining code s of the active process,
$Lvar : l$	the local state l of the active process, including method parameters and the implicit future identity destiny,
$Flds : a$	the state a of the fields, including class parameters,
$Cnt : n$	a counter n used to generate future identities,
$Mtd : m$	the name m of the current method.

Similarly, classes have the form

$$\mathbf{class}(Id : c, Par : z, Flds : a, Init : s, Mtds : q, Cnt : n)$$

where c is the class name, z the class parameters, a the fields, and s the initialization code. The variable q is a multiset of method definitions of the form

$$(m, \overline{p}, l, s)$$

where m is the method name, $\overline{p}$ is the list of parameters, l contains the local variables (including default values), and s is the code. The counter n in the class is used to generate object identities.

Messages have the form of invocation events as described above. And, a future unit is of the form $\mathbf{fut}(Id : u, Val : v)$ where u is the future identity and v is its value. The global history is represented by a unit $\mathbf{hist}(h)$ where h is finite sequence of events (initially empty). Remark that a system configuration contains exactly one history. The history is included to define the interleaving semantics upon which we derive our history-based reasoning formalism.

The initial state of an object o of class C with actual class parameter values $\overline{v}$ is denoted $init_{o:C(\overline{v})}$ and is defined by

$$init_{o:C(\overline{v})} \triangleq \mathbf{object}(Id : o, Cl : C, Pr : init_C, Lvar : \emptyset, Flds : a, Cnt : 0, Mtd : init)$$

where a is the initial state of the object fields given by $[\mathsf{this} \mapsto o, Par_C \mapsto \overline{v}, Flds_C \mapsto \overline{d}]$. Here Par_C, $Flds_C$, and $init_C$, represent the class parameters, the fields, and the initialization code of C, respectively. The class parameters Par_C are initialized by the actual parameters $\overline{v}$, the fields $Flds_C$ are initialized by default values $\overline{d}$ (of the appropriate types), and the initial code is ready to be executed with an empty local state.

A system is given by a set of self-contained classes $\overline{Cl}$, including a class $Main$, without class parameters, used to generate the initial object $init_{main:Main(\varepsilon)}$. The initial configuration of a system is defined by

$$init_{\overline{Cl}} \triangleq \overline{Cl} \;\; init_{main:Main(\varepsilon)} \;\; \mathbf{hist}(\varepsilon)$$

The operational rules are summarized in Fig. 4. The rules for skip, assignment, initialized variable declarations, if- and while-statements are standard. Note that $(a;\ l)$ represents the total object state, composed by a, the state of the fields/class parameters, and l, the state of the local variables/parameters of the method. Lookup of a variable if left to right, i.e., l is tried before a. Expressions e without side-effects are evaluated by a semantic function depending on the total state, i.e., $eval(e, (a;\ l))$.

Method invocation is captured by the rule call. The generated future identity $ft(o, n)$ is globally unique (assuming the *next* function is producing locally unique values). The future unit itself is not generated yet; it will be generated by return from the called method.

If there is no active process in an object, denoted $Pr : empty$, a method call is selected for execution by rule method. The invocation message is consumed by this rule, and the future identity of the call is assigned to the implicit parameter destiny. Method execution is completed by rule return, and a future value is fetched by rule query. A query can only succeed if the appropriate future unit is generated. A future unit appears in the configuration when resolved by rule return, which means that a query statement blocks until the future is resolved. Remark that rule query does not remove the future unit from the configuration, which allows several processes to fetch the value of the same future.

In rule new, the new object gets a unique identity, $ob(C, n)$, given by that of the generating object and a counter, the actual class parameters are evaluated, and the initialization is performed. The given language fragment may be extended with constructs for inter object process control and suspension, e.g., by using the *ABS* approach of [DDJO12].

$$
\begin{array}{ll}
\mathsf{skip:} & \mathbf{object}(Id: o, Pr:\ (\texttt{\textbf{skip}}; s)) \longrightarrow \mathbf{object}(Id: o, Pr: s) \\[2ex]
\mathsf{assign:} & \mathbf{object}(Id: o, Pr: (v := e; s), Lvar: l, Flds: a) \\
& \longrightarrow \\
& \texttt{\textbf{if}}\ \ v\ in\ l\ \texttt{\textbf{then}}\ \mathbf{object}(Id: o, Pr: s, Lvar: l[v \mapsto eval(e, (a;l))], Flds: a) \\
& \qquad\qquad \texttt{\textbf{else}}\ \mathbf{object}(Id: o, Pr: s, Lvar: l, Flds: a[v \mapsto eval(e, (a;l))]) \\[2ex]
\mathsf{init:} & \mathbf{object}(Id: o, Pr: (\overline{Tv := e}; s), Lvar: l, Flds: a) \\
& \longrightarrow \\
& \mathbf{object}(Id: o, Pr: (\overline{Tv}; \overline{v := e}; s), Lvar: l, Flds: a) \\[2ex]
\mathsf{if\text{-}else:} & \mathbf{object}(Id: o, Pr: (if\ \ e\ then\ s_1\ else\ s_2\ fi; s), Lvar: l, Flds: a) \\
& \longrightarrow \\
& \texttt{\textbf{if}}\ \ eval(e, (a;l))\ \texttt{\textbf{then}}\ \mathbf{object}(Id: o, Pr: (s_1; s), Lvar: l, Flds: a) \\
& \qquad\qquad\qquad\quad \texttt{\textbf{else}}\ \mathbf{object}(Id: o, Pr: (s_2; s), Lvar: l, Flds: a) \\[2ex]
\mathsf{while:} & \mathbf{object}(Id: o, Pr: (while\ e\ do\ s_1\ od; s), Lvar: l, Flds: a) \\
& \longrightarrow \\
& \mathbf{object}(Id: o, Pr: (if\ \ e\ then\ s_1; while\ e\ do\ s_1\ od\ fi; s), Lvar: l, Flds: a) \\[2ex]
\mathsf{new:} & \mathbf{hist}(h)\ \mathbf{class}(Id: C, Cnt: n) \\
& \mathbf{object}(Id: o, Pr: (v := new\ C(\overline{e}); s), Lvar: l, Flds: a) \\
& \longrightarrow \\
& \mathbf{hist}(h \cdot \langle o \xrightarrow{new} ob(C,n), C, eval(\overline{e}, (a;l))\rangle)\ \mathbf{class}(Id: C, Cnt: next(n)) \\
& \mathbf{object}(Id: o, Pr: (v := ob(C,n); s), Lvar: l, Flds: a) \\
& init_{ob(C,n):C(eval(\overline{e},(a;l)))} \\[2ex]
\mathsf{call:} & \mathbf{hist}(h)\ \mathbf{object}(Id: o, Pr: (fr := v!m(\overline{e}); s), Lvar: l, Flds: a, Cnt: n) \\
& \longrightarrow \\
& \text{MSG}\ \mathbf{hist}(h \cdot \text{MSG}) \\
& \mathbf{object}(Id: o, Pr: (fr := ft(o,n); s), Lvar: l, Flds: a, Cnt: next(n)) \\[2ex]
\mathsf{method:} & \langle o' \rightarrow o, u, m, \overline{v}\rangle\ \mathbf{hist}(h)\ \mathbf{class}(Id: c, Mtds: (q\ \ (m, \overline{p}, l, s))) \\
& \mathbf{object}(Id: o, Cl: c, Pr: empty, Flds: a) \\
& \longrightarrow \\
& \mathbf{hist}(h \cdot \langle o' \twoheadrightarrow o, u, m, \overline{v}\rangle)\ \mathbf{class}(Id: c, Mtds: (q\ \ (m, \overline{p}, l, s))) \\
& \mathbf{object}(Id: o, Cl: c, Pr: s, Lvar: l[\overline{p} \mapsto \overline{v}][\mathsf{destiny} \mapsto u], Flds: a, Mtd: m) \\[2ex]
\mathsf{return:} & \mathbf{hist}(h)\ \mathbf{object}(Id: o, Pr: \texttt{\textbf{return}}\ e, Lvar: l, Flds: a, Mtd: m) \\
& \longrightarrow \\
& \mathbf{hist}(h \cdot \langle \leftarrow o, eval(\mathsf{destiny}, l), m, eval(e, (a;l))\rangle) \\
& \mathbf{fut}(Id: eval(\mathit{destiny},l), Val: eval(e,(a;l))) \\
& \mathbf{object}(Id: o, Pr: empty, Flds: a) \\[2ex]
\mathsf{query:} & \mathbf{hist}(h)\ \mathbf{fut}(Id: u, Val: d)\ \mathbf{object}(Id: o, Pr: (v := e?; s), Lvar: l, Flds: a) \\
& \longrightarrow \\
& \mathbf{hist}(h \cdot \langle o \twoheadleftarrow, u, d\rangle)\ \mathbf{fut}(Id: u, Val: d) \\
& \mathbf{object}(Id: o, Pr: (v := d; s), Lvar: l, Flds: a) \\
& \texttt{\textbf{if}}\ eval(e, (a;l)) = u
\end{array}
$$

Figure 4. Operational rules, using the standard rewriting logic convention that irrelevant attributes may be omitted in a rule. Variables are denoted by single characters (the uniform naming convention is left implicit), $(a;\ l)$ represents the total object state, and $a[v \mapsto d]$ is the state a updated by binding the variable v to the data value d. The *eval* function evaluates an expression in a given state, and *in* is used for testing domain membership. In rule call, MSG denotes $\langle o \rightarrow eval(v, (a;\ l)), ft(o, n), m, eval(\overline{e}, (a;\ l))\rangle$

4.2. Semantic properties

Semantic properties are stated by means of notions of validity. We define *global validity* (denoted $\models$) and *local validity* with respect to a class C (denoted $\models_C$). A global object system initiated by a configuration $init_{\overline{Cl}}$ is said to satisfy a global invariant property $I(h)$, if the global history h of any reachable configuration G satisfies $I(h)$:

$$\overline{Cl} \models I(h) \triangleq \forall\, G \,.\, init_{\overline{Cl}} \longrightarrow^{*} G \wedge G.hist = h \Rightarrow I(h)$$

where $\longrightarrow^{*}$ denotes the transitive and reflexive extension of the transition relation, lifted to configurations, and where $G.hist$ extracts the history of the configuration G.

Similarly, an object system initiated by a configuration $init_{\overline{Cl}}$ is said to satisfy a C-local invariant property $I(h)$ if every object o of class C in any reachable configuration G satisfies $I(h/o)$, i.e., the projection from global history to the object o:

$$\overline{Cl} \models_C I(h) \triangleq \forall\, G, o \,.\, init_{\overline{Cl}} \longrightarrow^{*} G \wedge G.hist = h \wedge o \in G.obj \wedge G[o].class = C \Rightarrow I(h/o)$$

where $G.obj$ extracts the object identities from the objects in the configuration G.

We next provide notions of global and local well-formedness for global histories. We first introduce some notation and functions used in defining wellformed histories. For sequences a and b, let a **ew** x denote that x is the last element of a, $agree(a)$ denote that all elements (if any) are equal, and $a \leqslant b$ denote that a is a prefix of b. Let $[x_1, x_2, \ldots, x_i]$ denote the sequence of $x_1, x_2, \ldots, x_i$ for $i > 0$ (allowing repeated parts $[...]^*$). Functions for event decomposition are lifted to sequences in the standard way, ignoring events for which the decomposition is not defined, e.g., $\_.result : \mathsf{Seq[Ev]} \rightarrow \mathsf{Seq[Data]}$.

Functions may extract information from the history. In particular, we define $oid : \mathsf{Seq[Ev]} \rightarrow \mathsf{Set[Obj]}$ extracting all object identities occurring in a history, as follows:

$$\begin{array}{ll}
oid(\varepsilon) \triangleq \{main\} & oid(h \cdot \gamma) \triangleq oid(h) \cup oid(\gamma) \\
oid(\langle o \rightarrow o', u, m, (\overline{e})\rangle) \triangleq \{o, o'\} \cup oid(\overline{e}) & oid(\langle o' \twoheadrightarrow o, u, m, \overline{e}\rangle) \triangleq \{o, o'\} \cup oid(\overline{e}) \\
oid(\langle \leftarrow o, u, m, e\rangle) \triangleq \{o\} \cup oid(e) & oid(\langle o \twoheadleftarrow, u, e\rangle) \triangleq \{o\} \cup oid(e) \\
oid(\langle o \overset{new}{\longrightarrow} o', C, \overline{e}\rangle) \triangleq \{o, o'\} \cup oid(\overline{e}) &
\end{array}$$

where $\gamma : \mathsf{Ev}$, and $oid(\overline{e})$ returns the set of object identifiers occurring in the expression list $\overline{e}$. The function $fid : \mathsf{Seq[Ev]} \rightarrow \mathsf{Set[Fid]}$ extracts future identities from a history:

$$\begin{array}{ll}
fid(\varepsilon) \triangleq \emptyset & fid(h \cdot \gamma) \triangleq fid(h) \cup fid(\gamma) \\
fid(\langle o \rightarrow o', u, m, \overline{e}\rangle) \triangleq \{u\} & fid(\langle o' \twoheadrightarrow o, u, m, \overline{e}\rangle) \triangleq \{u\} \cup fid(\overline{e}) \\
fid(\langle \leftarrow o, u, m, e\rangle) \triangleq \emptyset & fid(\langle o \twoheadleftarrow, u, e\rangle) \triangleq fid(e) \\
fid(\langle o \overset{new}{\longrightarrow} o', C, \overline{e}\rangle) \triangleq fid(\overline{e}) &
\end{array}$$

where $\gamma : \mathsf{Ev}$, and $fid(\overline{e})$ returns the set of future identities occurring in the expression list $\overline{e}$. For a global history h, the function $fid(h)$ returns all future identities on h, and for a local history h/o, the function $fid(h/o)$ returns the futures generated by o or received as parameters. At last, h/u abbreviates the projection of history h to the set $\{\gamma \mid \gamma.future = u\}$, i.e., all events with future u.

Definition 4.1 (*Wellformed histories*) Let $h : \mathsf{Seq[Ev]}$ be a history of a global object system S. The well-formedness predicate $wf : \mathsf{Seq[Ev]} \rightarrow \mathsf{Bool}$ is defined by:

$$\begin{array}{ll}
wf(\varepsilon) & \triangleq true \\
wf(h \cdot \langle o \rightarrow o', u, m, \overline{e}\rangle) & \triangleq wf(h) \wedge o \neq \mathsf{null} \wedge u \notin fid(h) \cup fid(\overline{e}) \\
wf(h \cdot \langle o' \twoheadrightarrow o, u, m, \overline{e}\rangle) & \triangleq wf(h) \wedge o \neq \mathsf{null} \wedge h/u = [\langle o' \rightarrow o, u, m, \overline{e}\rangle] \\
wf(h \cdot \langle \leftarrow o, u, m, e\rangle) & \triangleq wf(h) \wedge h/u \;\mathbf{ew}\; \langle \_ \twoheadrightarrow o, u, m, \_\rangle \\
wf(h \cdot \langle o \twoheadleftarrow, u, e\rangle) & \triangleq wf(h) \wedge u \in fid(h/o) \wedge agree(((h/u).result) \cdot e) \\
wf(h \cdot \langle o \overset{new}{\longrightarrow} o', C, \overline{e}\rangle) & \triangleq wf(h) \wedge o \neq \mathsf{null} \wedge o' \neq \mathsf{null} \wedge o' \notin oid(h) \cup oid(\overline{e})
\end{array}$$

It follows directly that a wellformed global history satisfies the communication order pictured in Fig. 3, i.e.,

$$\begin{array}{l}
\forall\, u \,.\, \exists\, o, o', m, \overline{e}, e \,. \\
\qquad h/u \leqslant [\langle o' \rightarrow o, u, m, \overline{e}\rangle, \langle o' \twoheadrightarrow o, u, m, \overline{e}\rangle, \langle \leftarrow o, u, m, e\rangle, [\langle \_ \twoheadleftarrow, u, e\rangle]^*]
\end{array}$$

Also, it ensures the uniqueness of object identifiers and future identities. We can prove that the operational semantics guarantees well-formedness:

Lemma 4.1 *The global history h of a global object system S obtained by the given operational semantics, is wellformed, i.e.,* $\models wf(h)$ *where* $wf(h)$ *is strengthened by the two conditions* $fid(h) \subseteq (h/\ \rightarrow).future$ *and* $oid(h) - \mathsf{null} \subseteq (h/\ \xrightarrow{new}).callee$.

The two conditions ensure that a history may not refer to object and future identities before generated by creation and invocation events, respectively. This lemma follows by induction over the number of rule applications.

Well-formedness of a local history for an object o, denoted $wf_o(h)$, is defined as in Definition 4.1, except that the last conjunct of the case $\langle o' \twoheadrightarrow o, u, m, \overline{e}\rangle$ only holds for self calls, i.e., where o and o' are equal. For local well-formedness, the conjunct is therefore weakened to $o = o' \Rightarrow h/u = [\langle o' \rightarrow o, u, m, \overline{e}\rangle]$. If h is a wellformed global history, it follows immediately that each projection h/o is locally wellformed, i.e.,

$$wf(h) \Rightarrow wf_o(h/o)$$

5. Program verification

The communication history abstractly captures the system state at any point in time [Dah87, Dah92]. Partial correctness properties of a system may thereby be specified by finite initial segments of its communication histories. A *history invariant* is a predicate over the communication history, which holds for all finite sequences in the (prefix-closed) set of possible histories, expressing safety properties [AS85]. In this section we present a framework for compositional reasoning about object systems, establishing an invariant over the global history from invariants over the local histories of each object. Since the local object histories are disjoint with our four event semantics, it is possible to reason locally about each object. In particular, the history updates of the operational semantics affect the local history of the active object only, and can be treated simply as an assignment to the local history. The local history is not effected by the environment, and interference-free reasoning is then possible. Correspondingly, the reasoning framework consists of two parts: A proof system for local (class-based) reasoning, and a rule for composition of object specifications.

5.1. Local reasoning

Pre- and postconditions to method definitions are in our setting used to establish a *class invariant*. The class invariant must hold after initialization of all class instances and must be maintained by all methods, serving as a contract for the different methods: A method implements its part of the contract by ensuring that the invariant holds upon termination, assuming that it holds when the method starts execution. A class invariant establishes a *relationship between the internal state and the observable behavior of class instances*. The internal state reflects the values of the fields, and the observable behavior is expressed as potential communication histories. A *user-provided invariant* $I(\overline{w}, \mathcal{H})$ for a class C is a predicate over the fields $\overline{w}$, the read-only parameters $\overline{cp}$ and this, in addition to the local history $\mathcal{H}$ which is a sequence of events generated by this. The proof system for class-based verification is formulated within *dynamic logic* as used by the KeY framework [BHS07], facilitating class invariant verification by considering each method independently. The dynamic logic formulation suggests that the proof system is suitable for an implementation in the KeY framework.

Dynamic logic provides a structured way to describe program behavior by an integration of programs and assertions within a single language. The formula $[s]\phi$ expresses the precondition of s with ϕ as postcondition. The formula $\psi \Rightarrow [s]\phi$ express partial correctness properties: if statement s is executed in a state where ψ holds and the execution terminates, then ϕ holds in the final state. The formula is verified by a symbolic execution of s, where state modifications are handled by the *update* mechanism [BHS07]. A dynamic formula $[s_1;\ s]\phi$ is equal to $[s_1][s]\phi$. A dynamic formula $[v := e;\ s]\phi$, i.e., where an assignment is the first statement, reduces to $\{v := e\}[s]\phi$, where $\{v := e\}$ is an update. We assume that expressions e can be evaluated within the assertion language. Updates can only be *applied* on formulas without programs, which means that updates on a formula $[s]\phi$ are accumulated and *delayed* until the symbolic execution of s is complete. Update application $\{v := t\}e$, on an expression e, evaluates to the substitution e^v_t, replacing all free occurrences of v in e by t.

$$\text{invoc}\ \frac{\vdash \forall u\,.\,\{\mathcal{H} := \mathcal{H} \cdot \langle \mathsf{this} \rightarrow v, u, m, \overline{e}\rangle || \ fr := u\}\ [s]\phi}{\vdash [fr := v!m(\overline{e});\ \ s]\phi}$$

$$\text{fetch}\ \frac{\vdash \forall v'\,.\,\{\mathcal{H} := \mathcal{H} \cdot \langle \mathsf{this} \twoheadleftarrow, e, v'\rangle || \ v := v'\}\ ([s]\phi \wedge \exists \overline{w}\,.\, I(\overline{w}, \mathcal{H}))}{\vdash [v := e?;\ \ s]\phi}$$

$$\text{new}\ \frac{\vdash \forall v'\,.\,\{\mathcal{H} := \mathcal{H} \cdot \langle \mathsf{this} \overset{new}{\longrightarrow} v', C, \overline{e}\rangle || \ v := v'\}\ [s]\phi}{\vdash [v\ \ := \mathbf{new}\ C(\overline{e});\ \ s]\phi}$$

Figure 5. Dynamic logic rules for method invocation, future query and object creation. $I(\overline{w}, \mathcal{H})$ is the class invariant

skip	$[\mathbf{skip};\ \ s]\phi = [s]\phi$
assign	$[v := e;\ \ s]\phi = \{v := e\}\ [s]\phi$
declInit	$[T\ v = e;\ \ s]\phi = [v' := e;\ \ s^{v}_{v'}]\phi$
declNoInit	$[T\ v;\ \ s]\phi = [v' := \mathsf{default}_T;\ \ s^{v}_{v'}]\phi$
ifElse	$[\mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{fi};\ \ s]\phi = \mathbf{if}\ b\ \mathbf{then}\ [s_1; s]\phi\ \mathbf{else}\ [s_2; s]\phi$
while	$[\mathbf{while}\ b\ \mathbf{do}\ s'\ \mathbf{od};\ \ s]\phi = \mathbf{if}\ b\ \mathbf{then}\ (\exists w\,.\, I(\overline{w}, \mathcal{H}))\wedge$ $([s';\ \ \mathbf{while}\ b\ \mathbf{do}\ s'\ \mathbf{od};\ \ s]\phi)\ \mathbf{else}\ [s]\phi$

Figure 6. Semantical definitions for standard *ABS* statements. Here ϕ is the postcondition, s is the remaining program yet to be executed, primes denote fresh variables, $s^{v}_{v'}$ is s with all (free) occurrences of v replaced by v', and $\mathsf{default}_T$ is the default value defined for type T

The parallel update $\{v_1 := e_1\ ||\ ...\ ||\ v_n := e_n\}$, for disjoint variables $v_1, \ldots, v_n$, represents an accumulated update, and the application of a parallel update leads to a simultaneous substitution. For an update U, we have $U(\phi_1 \wedge \phi_2) = U\phi_1 \wedge U\phi_2$. A *sequent* $\psi_1, \ldots, \psi_n \vdash \phi_1, \ldots, \phi_m$ contains assumptions $\psi_1, \ldots, \psi_n$, and formulas $\phi_1, \ldots, \phi_m$ to be proved. The sequent is *valid* if at least one formula ϕ_i follows from the assumptions, and it can be interpreted as $\psi_1 \wedge ... \wedge \psi_n \Rightarrow \phi_1 \vee ... \vee \phi_m$.

In order to verify a class invariant $I(\overline{w}, \mathcal{H})$, we must prove that the invariant is established by the initialization code and maintained by all method definitions in C, assuming well-formedness of the local history. For a method definition $m(\overline{x})\{s;\ \ \mathbf{return}\ e\}$ in C, this amounts to a proof of the sequent:

$$\begin{aligned}\vdash (wf_{\mathit{this}}(\mathcal{H}) \wedge I(\overline{w}, \mathcal{H}) \Rightarrow [&\mathcal{H} := \mathcal{H} \cdot \langle \mathsf{caller} \twoheadrightarrow \mathsf{this}, \mathsf{destiny}, m, \overline{x}\rangle;\\ &s;\ \ \mathcal{H} := \mathcal{H} \cdot \langle \leftarrow \mathsf{this}, \mathsf{destiny}, m, e\rangle](wf_{\mathit{this}}(\mathcal{H}) \Rightarrow I(\overline{w}, \mathcal{H}))\end{aligned}$$

Here, the method body is extended with a statement for extending the history with the invocation reaction event, and the **return** statement is treated as a history extension. Dynamic logic rules for method invocation, future query, and object creation, can be found in Fig. 5. When invoking a method, the update in the premise of rule invoc captures the history extension and the generation of a fresh future identity u. Similarly, the update in rule fetch captures the history extension and the assignment of a fresh value to v, where the well-formedness assumptions ensure that all values received from the same future are equal. The update in the premise of rule new captures the history extension and the generation of a fresh object identity v', and the universal quantifier reflects non-determinism. The prime is needed here since v may occur in $\overline{e}$. The query rule insists that the class invariant holds for local history, ignoring the field values of the current state, as discussed in the soundness proof. Assignments are analyzed as explained above, and rules for **skip** and conditionals are standard. We refer to Din et al. for further details [DDO12a].

The rules for the rest of the *ABS* statements can be defined as substitution rules introduced in Fig. 6. For instance, $[\mathbf{skip};\ \ s]\phi$ can be rewritten to $[s]\phi$. In rule declInit and declNoInit v' is needed since the postcondition may talk about a field with the same name v. If-statements without an else-branch are as usual.

5.2. Soundness

The reasoning system for statements in dynamic logic is sound if any provable property is valid, i.e.,

$$\vdash \psi \Rightarrow [s]\phi \Rightarrow \models \psi \Rightarrow [s]\phi$$

Validity of a dynamic logic formula, denoted $\models \psi \Rightarrow [s]\phi$, is defined by means of the operational semantics. We base the semantics on the operational semantics above, as given by unlabeled transitions of the form $G_1 \rightarrow G_2$.

Note that each rule is local to one object, and we write $G_1 \xrightarrow{o:s} G_2$ to indicate an execution involving only object o such that exactly the statement (list) s has been executed by o. And we write $G_1 \overset{o:s}{\rightrightarrows} G_2$ if o executes s while other objects may execute.

Definition 5.1 (*Explicit execution step*)

$$G_1 \overset{o:s}{\rightrightarrows} G_2 \triangleq G_1 \longrightarrow^* G_2 \wedge G_1[o].Pr = s;\ G_2[o].Pr$$

$$G_1 \xrightarrow{o:s} G_2 \triangleq G_1 \overset{o:s}{\rightrightarrows} G_2 \wedge \forall\, o'\,.\, o' \neq o \Rightarrow G_1[o'] = G_2[o']$$

expressing one or more transitions from the configuration G_1 to G_2 such that o executes s, with or without, respectively, interleaved execution by other objects. The notation $G[o]$ denotes the object o of the configuration G.

We consider pre- and postconditions over local states and the local history. Such an assertion can be evaluated in a state defining values for attributes (of the appropriate class), parameters and local variables (of the method) and the local history. We let $[\![\psi \xrightarrow{o:s} \phi]\!]_{G,o}$ express that if the condition ψ holds for object o before execution of s by the object in configuration G, then ϕ holds for o after the execution. As above, we let $\xrightarrow{o:s}$ express local execution by o, and $\overset{o:s}{\rightrightarrows}$ execution by o interleaved with other objects:

Definition 5.2 (*Validity of pre/post-conditions over execution steps*)

$$[\![\psi \xrightarrow{o:s} \phi]\!]_{G,o} \triangleq \forall G', \overline{z}\,.\, wf(G'.hist) \wedge G \xrightarrow{o:s} G' \wedge loc(G, o)[\psi] \Rightarrow loc(G', o)[\phi]$$
$$[\![\psi \overset{o:s}{\rightrightarrows} \phi]\!]_{G,o} \triangleq \forall G', \overline{z}\,.\, wf(G'.hist) \wedge G \overset{o:s}{\rightrightarrows} G' \wedge loc(G, o)[\psi] \Rightarrow loc(G', o)[\phi]$$

where $\overline{z}$ is the list of auxiliary variables in ψ and/or ϕ, not bound by G nor G'. Here $loc(G, o)$ denotes the local state of object o, as derived from the global state G. The function loc : $\mathsf{Config} \times \mathsf{Oid} \rightarrow \mathsf{State}$ is defined by

$$loc(G, o) \triangleq (G[o].Flds;\ G[o].Lvar) + [\mathcal{H} \mapsto (G.hist)/o]$$

where the resulting $\mathcal{H}$ ranges over local histories (i.e., in the alphabet of o), and where this is bound to o in G as explained earlier. Thus the extraction is made by taking the state of object o and adding the history localized to o. We let $loc(G, o)[\psi]$ denote the value of ψ in state $loc(G, o)$.

It follows that local reasoning suffices for local pre/post-conditions, in the sense that when reasoning about one object in our system, one may ignore the activity of other objects.

Lemma 5.1 $[\![\psi \xrightarrow{o:s} \phi]\!]_{G,o}$ *is the same as* $[\![\psi \overset{o:s}{\rightrightarrows} \phi]\!]_{G,o}$

The lemma follows by induction on the length of executions, and the fact that $loc(G, o)$ for any G is not affected by execution steps by other objects than o, since remote access to fields is not allowed in our language and since h/o only contains events generated by o.

In our setting, we may understand a sequent by means of the $\xrightarrow{o:s}$ relation, letting a dynamic logic subformula depend on a given pre-configuration G and object o.

Definition 5.3 (*Validity of dynamic logic sequents*)

$$
\begin{aligned}
\models \psi_1, \ldots, \psi_n \vdash \phi_1, \ldots, \phi_m &\triangleq \forall G, o \,.\, wf(G.hist) \Rightarrow [\![\psi_1 \wedge \ldots \wedge \psi_n \Rightarrow \phi_1 \vee \ldots \vee \phi_m]\!]_{G,o} \\
[\![[s]\phi]\!]_{G,o} &\triangleq [\![true \xrightarrow{o:s} \phi]\!]_{G,o} \\
[\![e]\!]_{G,o} &\triangleq loc(G, o)[e] \\
[\![\{\overline{v := t}\}e]\!]_{G,o} &\triangleq [\![e^{\overline{v}}_{\overline{t}}]\!]_{G,o} \\
[\![\psi \wedge \phi]\!]_{G,o} &\triangleq [\![\psi]\!]_{G,o} \wedge [\![\phi]\!]_{G,o} \\
[\![\psi \vee \phi]\!]_{G,o} &\triangleq [\![\psi]\!]_{G,o} \vee [\![\phi]\!]_{G,o} \\
[\![\psi \Rightarrow \phi]\!]_{G,o} &\triangleq [\![\psi]\!]_{G,o} \Rightarrow [\![\phi]\!]_{G,o} \\
[\![U(\psi \wedge \phi)]\!]_{G,o} &\triangleq [\![U\psi]\!]_{G,o} \wedge [\![U\phi]\!]_{G,o} \\
[\![U(\psi \vee \phi)]\!]_{G,o} &\triangleq [\![U\psi]\!]_{G,o} \vee [\![U\phi]\!]_{G,o} \\
[\![U(\psi \Rightarrow \phi)]\!]_{G,o} &\triangleq [\![U\psi]\!]_{G,o} \Rightarrow [\![U\phi]\!]_{G,o}
\end{aligned}
$$

where e is a formula without the dynamic logic operators, and the equations for updates are as given earlier. The application of a parallel update U, for instance, $\{v_1 := t_1 \mid\mid \ldots \mid\mid v_n := t_n\}$, is for short written as $\{\overline{v := t}\}$. It follows from the definition that

$$[\![\psi \Rightarrow [s]\phi]\!]_{G,o} = [\![\psi \xrightarrow{o:s} \phi]\!]_{G,o}$$

Here o is the executing object and the object on which ψ and ϕ are interpreted. Thus the formula is valid if for any object o executing s, the postcondition holds in the poststate, provided the precondition holds in the prestate. In dynamic logic the prestate given by G and o is fixed for the whole sequent, and therefore the meaning of the individual operators is given in the context of G and o.

We verify an invariant $I(\overline{w}, \mathcal{H})$ for a class C by showing that $I(\overline{w}, \mathcal{H})$ is established by the initialization of C, i.e. $init_C$, and is maintained by all methods in C, assuming local well-formedness. The rule is:

$$
\textsf{class}\ \frac{
\begin{array}{l}
\vdash \exists \overline{w} \,.\, I(\overline{w}, \mathcal{H} \cdot \gamma) \Rightarrow \exists \overline{w} \,.\, I(\overline{w}, \mathcal{H}) \\
\vdash \mathcal{H} = \varepsilon \Rightarrow [init_C](wf_{\textsf{this}}(\mathcal{H}) \Rightarrow I(\overline{w}, \mathcal{H})) \\
\vdash (wf_{\textsf{this}}(\mathcal{H}) \wedge I(\overline{w}, \mathcal{H})) \Rightarrow [body_{C,m}](wf_{\textsf{this}}(\mathcal{H}) \Rightarrow I(\overline{w}, \mathcal{H})), \text{ for all methods } m \text{ in } C
\end{array}
}{\vdash_C \exists \overline{w} \,.\, I(\overline{w}, \mathcal{H})}
$$

where $body_{C,m}$ denotes the body s of method m of C augmented with effects on the local history reflecting the start and end of the method, namely

$$\mathcal{H} := \mathcal{H} \cdot \langle \textsf{caller} \twoheadrightarrow \textsf{this}, \textsf{destiny}, m, \overline{x} \rangle;\ s;\ \ \mathcal{H} := \mathcal{H} \cdot \langle \leftarrow \textsf{this}, \textsf{destiny}, m, e \rangle$$

Lemma 5.2 *Reasoning about statements is sound:*

$$\vdash \psi \Rightarrow [s]\phi \Rightarrow \models \psi \Rightarrow [s]\phi$$

Theorem 5.1 *The proof system for reasoning about classes is sound:*

$$\vdash_C I(\mathcal{H}) \Rightarrow \models_C I(\mathcal{H})$$

Proof of Lemma 5.2. We focus on the rules for statements involving futures and object generation, and consider therefore the rules invoc, fetch and new, as given in Fig. 5. The axioms given in Fig. 6 represent standard statements not involving futures, and we omit the soundness proof of these. □

*5.2.1. **Asynchronous method call statement***

We prove that the invoc rule preserves validity. The validity of the conclusion is $\models [fr := v!m(\overline{e});\ \ s]\phi$. Consider now a given G and o, and let ϕ' denote $[s]\phi$. According to Definition 5.3, the validity can be written as

$$wf(G.hist) \Rightarrow [\![true \xrightarrow{o:fr:=v!m(\overline{e})} \phi']\!]_{G,o}$$

which by Definition 5.2 is

$$\forall\, G', \overline{z}\,.\, wf(G.hist) \wedge wf(G'.hist) \wedge G \xrightarrow{o:fr:=v!m(\overline{e})} G' \Rightarrow loc(G', o)[\phi']$$

By the operational semantics of call and assign, we have that G' is G with MSG and $G'.hist = G.hist \cdot$ MSG, where MSG denotes $\langle o \rightarrow loc(G, o)[v], ft(o, n), m, loc(G, o)[\overline{e}]\rangle$, and such that the object state $G'[o].l$ is $(G[o].l)[fr \mapsto ft(o, n)]$ if $fr \in G[o].l$, and otherwise $G'[o].a$ is $(G[o].a)[fr \mapsto ft(o, n)]$. Here n is the counter value of $G[o]$ (same as in $G'[o]$). Other parts of the object state are unchanged.

Thus $loc(G', o)[\phi']$ can be reduced to $loc(G, o)[\phi'\,{}^{fr, \mathcal{H}}_{ft(o,n), \mathcal{H}\cdot\langle \mathsf{this} \rightarrow v, ft(o,n), m, \overline{e}\rangle}]$ since $loc(G, o)[\langle \mathsf{this} \rightarrow v, ft(o, n), m, \overline{e}\rangle] =$ MSG, and it suffices to prove

$$\forall\, \overline{z}\,.\, wf(G.hist) \Rightarrow loc(G, o)[\phi'\,{}^{fr, \mathcal{H}}_{ft(o,n), \mathcal{H}\cdot\langle \mathsf{this} \rightarrow v, ft(o,n), m, \overline{e}\rangle}]$$

The validity of the premise is

$$\models \forall u\,.\, \{\mathcal{H} := \mathcal{H} \cdot \langle \mathsf{this} \rightarrow v, u, m, \overline{e}\rangle \;||\; fr := u\}\; \phi'$$

which by Definition 5.3 is

$$\forall\, \overline{z}, u\,.\, wf(G.hist) \Rightarrow loc(G, o)[\phi'\,{}^{fr, \mathcal{H}}_{u, \mathcal{H}\cdot\langle \mathsf{this} \rightarrow v, u, m, \overline{e}\rangle}]$$

Clearly this is sufficient to ensure validity of the conclusion, since the universal quantifier on u covers the value given by $ft(o, n)$.

5.2.2. *Query statement*

We prove that the fetch rule preserves validity. The validity of the conclusion is $\models [v := e?;\;\; s]\phi$. Consider now a given G and o, and let ϕ' denote $[s]\phi$. According to Definition 5.3, the validity can be written as

$$wf(G.hist) \Rightarrow [\![true \xrightarrow{o:v:=e?} \phi']\!]_{G,o}$$

which by Definition 5.2 is

$$\forall G', \overline{z}\,.\, wf(G.hist) \wedge wf(G'.hist) \wedge G \xrightarrow{o:v:=e?} G' \Rightarrow loc(G', o)[\phi']$$

By the operational semantics of query and assign, we have that G' is G with MSG and $G'.hist = G.hist \cdot$ MSG, where MSG denotes $\langle o \twoheadleftarrow, loc(G, o)[e], d\rangle$ and such that the object state $G'[o].l$ is $(G[o].l)[v \mapsto d]$ if $v \in G[o].l$, and otherwise $G'[o].a$ is $(G[o].a)[v \mapsto d]$. Other parts of the object state are unchanged.

Thus $loc(G', o)[\phi']$ can be reduced to $loc(G, o)[\phi'\,{}^{v, \mathcal{H}}_{d, \mathcal{H}\cdot\langle \mathsf{this} \twoheadleftarrow, e, d\rangle}]$ since $loc(G, o)[\langle \mathsf{this} \twoheadleftarrow, e, d\rangle] =$ MSG, and it suffices to prove

$$\forall\, \overline{z}\,.\, wf(G.hist) \Rightarrow loc(G, o)[\phi'\,{}^{v, \mathcal{H}}_{d, \mathcal{H}\cdot\langle \mathsf{this} \twoheadleftarrow, e, d\rangle}]$$

The validity of the premise is

$$\models \forall v'\,.\, \{\mathcal{H} := \mathcal{H} \cdot \langle \mathsf{this} \twoheadleftarrow, e, v'\rangle \;||\;\; v := v'\}\; (\phi' \wedge \exists\, \overline{w}\,.\, I(\overline{w}, \mathcal{H}))$$

which by Definition 5.3 is

$$\forall\, \overline{z}, v'\,.\, wf(G.hist) \Rightarrow loc(G, o)[(\phi' \wedge \exists\, \overline{w}\,.\, I(\overline{w}, \mathcal{H}))\,{}^{v, \mathcal{H}}_{v', \mathcal{H}\cdot\langle \mathsf{this} \twoheadleftarrow, e, v'\rangle}]$$

Clearly this is sufficient to ensure validity of the conclusion, since the universal quantifier on v' covers the value given by d. Note that the invariant is not required here.

5.2.3. *Object creation statement*

We prove that the new rule preserves validity. The validity of the conclusion is $\models [v := new\ C(\overline{e});\;\; s]\phi$. Consider now a given G and o, and let ϕ' denote $[s]\phi$. According to Definition 5.3, the validity can be written as

$$wf(G.hist) \Rightarrow [\![true \xrightarrow{o:v:=new\ C(\overline{e})} \phi']\!]_{G,o}$$

which by Definition 5.2 is

$$\forall G', \overline{z}\,.\, wf(G.hist) \wedge wf(G'.hist) \wedge G \xrightarrow{o:v:=new\ C(\overline{e})} G' \Rightarrow loc(G', o)[\phi']$$

By the operational semantics of new and assign, we have that G' is G with MSG and $G'.hist = G.hist \cdot$MSG, where MSG denotes $\langle o \xrightarrow{new} ob(C, n), C, loc(G, o)[\overline{e}]\rangle$ and such that the object state $G'[o].l$ is $(G[o].l)[v \mapsto ob(C, n)]$ if $v \in G[o].l$, and otherwise $G'[o].a$ is $(G[o].a)[v \mapsto ob(C, n)]$. Here n is the counter value of $G[C]$ (same as in $G'[C]$). Other parts of the object state are unchanged.

Thus $loc(G', o)[\phi']$ can be reduced to $loc(G, o)[\phi'\,{}^{v,\mathcal{H}}_{ob(C,n),\mathcal{H}\cdot\langle \mathsf{this} \xrightarrow{new} ob(C,n), C, \overline{e}\rangle}]$ since $loc(G, o)[\langle \mathsf{this} \xrightarrow{new} ob(C, n), C, \overline{e}\rangle] =$ MSG, and it suffices to prove

$$\forall \overline{z}\,.\, wf(G.hist) \Rightarrow loc(G, o)[\phi'\,{}^{v,\mathcal{H}}_{ob(C,n),\mathcal{H}\cdot\langle \mathsf{this} \xrightarrow{new} ob(C,n), C, \overline{e}\rangle}]$$

The validity of the premise is

$$\models \forall v'\,.\,\{\mathcal{H} := \mathcal{H} \cdot \langle \mathsf{this} \xrightarrow{new} v', C, \overline{e}\rangle \mid\mid v := v'\}\, \phi'$$

which by Definition 5.3 is

$$\forall \overline{z}, v'\,.\, wf(G.hist) \Rightarrow loc(G, o)[\phi'\,{}^{v,\mathcal{H}}_{v',\mathcal{H}\cdot\langle \mathsf{this} \xrightarrow{new} v', C, \overline{e}\rangle}]$$

Clearly this is sufficient to ensure validity of the conclusion, since the universal quantifier on v' covers the value given by $ob(C, n)$.

Proof of Theorem 5.1. The theorem follows by Lemma 5.2 above and by proving that if one can prove $\vdash_C I'(\mathcal{H})$ by the class rule, then $\models_C I'(\mathcal{H})$, letting $I'(\mathcal{H})$ denote $\exists\, \overline{w}\,.\, I(\overline{w}, \mathcal{H})$. $\square$

Consider the rule class. We may assume that the premises of the rule are valid. By definition, the validity of $I'(\mathcal{H})$ is

$$\forall G, o\,.\, init_{\overline{Cl}} \longrightarrow^{*} G \wedge G.hist = \mathcal{H} \wedge o \in G.obj \wedge G[o].class = C \Rightarrow I'(\mathcal{H}/o)$$

We first prove that this holds for all C objects o in states G such that $G[o].\mathrm{Pr} = empty$. With the given operational semantics, Pr is *empty* for an object o when o has finished a method, or $init_C$, and it can only start a new method when Pr is *empty*. By Lemma 4.1 we only need to consider states with a wellformed history. We need to show that the invariant $I(\overline{w}, \mathcal{H})$ holds after the initialization and is maintained by every methods of class C, considering any interleaved execution according to the operational semantics. The validity of the second premise gives

$$\forall G, o\,.\, wf(G.hist) \Rightarrow [\![\mathcal{H} = \varepsilon \xrightarrow{o:init_C} (wf_{\mathsf{this}}(\mathcal{H}) \Rightarrow I(\overline{w}, \mathcal{H}))]\!]_{G,o}$$

which by Lemma 5.1 is the same as

$$\forall G, o\,.\, wf(G.hist) \Rightarrow [\![\mathcal{H} = \varepsilon \overset{o:init_C}{\rightrightarrows} (wf_{\mathsf{this}}(\mathcal{H}) \Rightarrow I(\overline{w}, \mathcal{H}))]\!]_{G,o}$$

which by definition is

$$G \overset{o:init_C}{\rightrightarrows} G' \wedge loc(G, o)[\mathcal{H} = \varepsilon] \Rightarrow loc(G', o)[wf_{\mathsf{this}}(\mathcal{H}) \Rightarrow I(\overline{w}, \mathcal{H})]$$

for all states G' with wellformed histories, and $\overline{z}$. This states that the class invariant $I(\overline{w}, \mathcal{H})$ holds after the initialization of class C, conditioned by local well-formedness. The condition on local well-formedness follows from the global well-formedness $wf(G'.hist)$. The condition $loc(G, o)[\mathcal{H} = \varepsilon]$ follows by induction on the length of an execution showing that no object can generate events before its initial code has started.

Similarly, the validity of the third premise gives that

$$G \overset{o:body_{C,m}}{\rightrightarrows} G' \wedge loc(G, o)[wf_{\mathsf{this}}(\mathcal{H}) \wedge I(\overline{w}, \mathcal{H})] \Rightarrow loc(G', o)[wf_{\mathsf{this}}(\mathcal{H}) \Rightarrow I(\overline{w}, \mathcal{H})]$$

for all states G, G' with wellformed histories, and all o, $\overline{z}$. This states that the class invariant is maintained by a method m of C under the assumption of local well-formedness. As before local well-formedness follows from global well-formedness. Thus $I(\overline{w}, \mathcal{H})$, and therefore also $I'(\mathcal{H})$, hold for all C objects o in reachable states G with empty $G[o].\mathrm{Pr}$.

It remains to show that the invariant also holds in states G where $G[o]$. Pr is nonempty. By the first premise, we have that I' is prefix-closed with respect to the history. Thus all states in between those where $G[o]$. Pr is empty will also satisfy I'. In order to ensure I' in case of nonterminating methods (or *init*), we must consider loops and other sources of non-termination. For loops it suffices to let I' be required at the beginning of each loop iteration, which we do require in the while axiom. The other source of nonterminating methods is the query statement; however, here the proof rule fetch insists that we verify I'. Thus any proof of that method (or *init*) must establish I' at his point. By Lemma 5.2 we have that I' is valid. We may conclude that reasoning about classes is sound.

We remark that it would be sufficient to verify I' for queries where the caller of the future equals this as reasoning is local, and independent of the behavior of other objects. But this would require notation for expressing the caller of a future (say u.caller, defined by $ft(o, n)$.caller $= o$) in the specification (and possibly programming) language. However, the verification cost of having I' in the rule for query and in the axiom for *while*, is not great since one is obliged to prove $I(\overline{w}, \mathcal{H})$ at the end of the body.

5.3. Compositional reasoning

The class invariant $I(\overline{w}, \mathcal{H})$ for some class C is a predicate over the fields $\overline{w}$, the local history $\mathcal{H}$, as well as the formal class parameters $\overline{cp}$ and this, which are constant (read-only) variables. *History invariants* $I_C(\mathcal{H})$ for instances of C, expressed as a predicate over the local history, can be derived from the class invariant by hiding fields, i.e., $\exists\, \overline{w} \,.\, I(\overline{w}, \mathcal{H})$.

$$I_C(\mathcal{H}) \triangleq \exists\, \overline{w} \,.\, I(\overline{w}, \mathcal{H})$$

Notice that the history invariants should be prefix-closed since according to the definition in Sect. 4.2 C-local invariant property must be satisfied by all reachable states. Consequently, $\exists\, \overline{w} \,.\, I(\overline{w}, \mathcal{H})$ should be weakened if needed in order to obtain prefix-closedness. Then we assume from now on that $I_C(\mathcal{H})$ is prefix-closed.

For an instance o of C with actual parameter values $\overline{e}$, the *object invariant* $I_{o:C(\overline{e})}(h)$ is defined by the class invariant applied to the local projection of the history and instantiating this and the class parameters:

$$I_{o:C(\overline{e})}(h) \triangleq I_C(h/o)^{\mathsf{this}, \overline{cp}}_{o, \overline{e}}$$

where I_C is a prefix-closed class invariant as above, with hidden internal state $\overline{w}$. We consider a composition rule for a system S of objects $o : C(\overline{e})$ together with dynamically generated objects by S. The history invariant $I_S(h)$ for such a system is then given by combining the history invariants of the composed objects:

$$I_S(h) \triangleq wf(h) \bigwedge_{(o:C(\overline{e}))\in S\cup ob(h)} I_{o:C(\overline{e})}(h)$$

where the function ob : Seq[Ev] $\rightarrow$ Set[Obj $\times$ Cls $\times$ List[Data]] returns the set of created objects (each given by its object identity, associated class and class parameters) in a history:

$$\begin{array}{ll}
ob(\varepsilon) & \triangleq \{main : Main(\varepsilon)\} \\
ob(h \cdot \langle o \xrightarrow{new} o', C, \overline{e}\rangle) & \triangleq ob(h) \cup \{o' : C(\overline{e})\} \\
ob(h \cdot \boldsymbol{others}) & \triangleq ob(h)
\end{array}$$

(where ***others*** matches all other events). By choosing S as $\{main : Main(\varepsilon)\}$ we may reason about a global system by means of $I_{\{main:Main(\varepsilon)\}}(h)$.

The local histories represent the activity of each concurrent object. Each wellformed interleaving of the local histories represents a possible global history. The set of possible wellformed global histories reflects interleaving semantics where the relative speed of each object is non-deterministic. The concurrent objects may execute in true concurrency, but the events are interleaved when we consider the global system since they are considered timeless. Note that the system invariant is obtained directly from the history invariants of the composed objects, without any restrictions on the local reasoning, since the local histories are disjoint. This ensures compositional reasoning. The composition rule is similar to [DDJO12], which also considers dynamically created objects.

5.4. Soundness proof of compositional reasoning

The proof rule for composition is:

$$\text{composition}\ \frac{\vdash_C I_C(h),\ \text{for each C in } \overline{Cl}}{\overline{Cl} \vdash wf(h) \quad \bigwedge_{(o:C(\overline{e}))\in ob(h)} I_{o:C(\overline{e})}(h)}$$

Note that $\vdash_C I_C(h)$ is trivial for $I_C(h) \triangleq true$, thus one may provide invariants for a subset of the classes and using *true* as default invariant for the rest.

Theorem 5.2 *The object composition rule is sound.*

Proof of Theorem 5.2. We show that the composition rule preserves soundness. For each class C we may then assume $\models_C I_C(h)$ which by definition is

$$\forall G, o\,.\, init_{\overline{Cl}} \longrightarrow^* G \wedge G.hist = h \wedge o \in G.obj \wedge G[o].class = C \Rightarrow I_{o:C(\overline{e})}(h)$$

Next we prove $\models I_{o:C(\overline{e})}(h)$ for all C-objects in h, i.e.,

$$\forall G\,.\, init_{\overline{Cl}} \longrightarrow^* G \wedge G.hist = h \Rightarrow \bigwedge_{(o:C(\overline{e}))\in ob_C(h)} I_{o:C(\overline{e})}(h)$$

letting $ob_C(h)$ denote the set of all C-objects in h. This reduces to proving that each C-object in $G.hist$ is found in $G.obj$. This can be proved by induction on the length of an execution. Finally by Lemma 4.1 we have $\models wf(h)$; and since conjunction commutes with validity we have $\models wf(h) \quad \bigwedge_{(o:C(\overline{e}))\in ob(h)} I_{o:C(\overline{e})}(h)$. □

5.5. Example

In this example we consider object systems based on the classes found in Listings 1 and 2. In order to prove that MapReduce really does output the correct number of occurrences of each word in the collection of documents, each class should guarantee the corresponding functional correctness. For example, the method *invokeMap* does take a file and emit a list of pairs such that each word in the file is associated with a counting number "1". Moreover, we need to specify class invariants which capture the concurrent interaction between the Worker objects and the MapReduce object. For instance, the Reduce phase handled by the method *mapReduce* will start only after the Map phase has been completed. In this paper the functional correctness is given by assumption and we focus on the compositional proof based on histories such that we can derived the system property mentioned above.

Assume that the global system consists of the objects $w_1 : Worker$, $w_2 : Worker$, $w_3 : Worker$, $mr : MapReduce(wp)$, and $m : Main(mr)$, where the only visible activity of m is that it invokes $mapReduce$ method on the object mr. The semantics may lead to several global histories for this system, depending on the interleaving of the different object activities. For convenience, below we abbreviate the method names $mapReduce$ to mR, $invokeMap$ to ivM, and $invokeReduce$ to ivR. One global history h caused by a call to mR on mr from m is as follows:

$$\begin{aligned}
&[\langle m \to mr, u_1, \mathsf{mR}, \overline{e_1}\rangle, \langle m \twoheadrightarrow mr, u_1, \mathsf{mR}, \overline{e_1}\rangle,\\
&\langle mr \to w_1, u_2, \mathsf{ivM}, \overline{e_2}\rangle, \langle mr \to w_2, u_3, \mathsf{ivM}, \overline{e_3}\rangle,\\
&\langle mr \twoheadrightarrow w_2, u_3, \mathsf{ivM}, \overline{e_3}\rangle, \langle mr \twoheadrightarrow w_1, u_2, \mathsf{ivM}, \overline{e_2}\rangle,\\
&\langle \leftarrow w_1, u_2, \mathsf{ivM}, e_2\rangle, \langle \leftarrow w_2, u_3, \mathsf{ivM}, e_3\rangle, \langle mr \twoheadleftarrow, u_2, e_2\rangle, \langle mr \twoheadleftarrow, u_3, e_3\rangle,\\
&\langle mr \to w_2, u_4, \mathsf{ivR}, \overline{e_4}\rangle, \langle mr \twoheadrightarrow w_2, u_4, \mathsf{ivR}, \overline{e_4}\rangle, \langle mr \to w_1, u_5, \mathsf{ivR}, \overline{e_5}\rangle,\\
&\langle mr \to w_3, u_6, \mathsf{ivR}, \overline{e_6}\rangle, \langle mr \twoheadrightarrow w_3, u_6, \mathsf{ivR}, \overline{e_6}\rangle, \langle \leftarrow w_2, u_4, \mathsf{ivR}, e_4\rangle,\\
&\langle \leftarrow w_3, u_6, \mathsf{ivR}, e_6\rangle, \langle mr \twoheadrightarrow w_1, u_5, \mathsf{ivR}, \overline{e_5}\rangle, \langle \leftarrow w_1, u_5, \mathsf{ivR}, e_5\rangle,\\
&\langle mr \twoheadleftarrow, u_4, e_4\rangle, \langle mr \twoheadleftarrow, u_6, e_6\rangle, \langle mr \twoheadleftarrow, u_5, e_5\rangle, \langle \leftarrow mr, u_1, \mathsf{mR}, e_1\rangle]
\end{aligned}$$

It follows that the Reduce phase will starts only after the Map phase has been completed. In addition, none of the requests sent out to the workers is uncompleted when the call to mR on mr is finished. We may derive these properties within the proof system from the following class invariants:

$$
\begin{aligned}
&I_{Worker}(\mathcal{H}) \triangleq \mathcal{H} \leqslant [\langle c \twoheadrightarrow \mathsf{this}, u_1, \mathsf{ivM}, \overline{e_1}\rangle, \langle \leftarrow \mathsf{this}, u_1, \mathsf{ivM}, e_1\rangle \mid \\
&\quad \langle c \twoheadrightarrow \mathsf{this}, u_2, \mathsf{ivR}, \overline{e_2}\rangle, \langle \leftarrow \mathsf{this}, u_2, \mathsf{ivR}, e_2\rangle \,.\, \mathbf{some}\ c, u_1, u_2, \overline{e_1}, e_1, \overline{e_2}, e_2]^* \\
&I_{MapReduce(wp)}(\mathcal{H}) \triangleq \mathcal{H} \leqslant [\langle c \twoheadrightarrow \mathsf{this}, d, \mathsf{mR}, \overline{e_1}\rangle, \langle \mathsf{this} \rightarrow \_, \_, \mathsf{ivM}, \_\rangle^a, \\
&\qquad\qquad \langle \mathsf{this} \twoheadleftarrow, \_, \_\rangle^a, \langle \mathsf{this} \rightarrow \_, \_, \mathsf{ivR}, \_\rangle^b, \langle \mathsf{this} \twoheadleftarrow, \_, \_\rangle^b, \\
&\qquad\qquad\qquad \langle \leftarrow \mathsf{this}, d, \mathsf{mR}, e_1\rangle \,.\, \mathbf{some}\ c, d]^*
\end{aligned}
$$

Here we use regular expression notation to express patterns over the history, letting $\mid$ denote choice, letting superscript b specify b repetitions of a pattern, and $h \leqslant p^*$ express that h is a prefix of a repeated pattern p where additional variables occurring in p (after **some**) may change for each repetition. Notice that the class invariant of $MapReduce$ ensures that for each of the invocation event in $\langle \mathsf{this} \rightarrow \_, \_, \mathsf{ivM}, \_\rangle^a$, there is a corresponding fetch event in $\langle \mathsf{this} \twoheadleftarrow, \_, \_\rangle^a$ by the same future identity. Same approach is applied to $\langle \mathsf{this} \rightarrow \_, \_, \mathsf{ivR}, \_\rangle^b$ and $\langle \mathsf{this} \twoheadleftarrow, \_, \_\rangle^b$. These class invariants are straightforwardly verified in the above proof system.

The corresponding object invariants for $w_1 : Worker$, $w_2 : Worker$, $w_3 : Worker$ and $mr : MapReduce(wp)$ are obtained by substituting actual values for this and class parameters:

$$
\begin{aligned}
&I_{w_i:Worker}(h) \triangleq h/w_i \leqslant [\langle \_ \twoheadrightarrow w_i, u_1, \mathsf{ivM}, \overline{e_1}\rangle, \langle \leftarrow w_i, u_1, \mathsf{ivM}, e_1\rangle \mid \\
&\quad \langle \_ \twoheadrightarrow w_i, u_2, \mathsf{ivR}, \overline{e_2}\rangle, \langle \leftarrow w_i, u_2, \mathsf{ivR}, e_2\rangle \,.\, \mathbf{some}\ u_1, u_2, \overline{e_1}, e_1, \overline{e_2}, e_2]^* \\
&I_{mr:MapReduce(wp)}(h) \triangleq h/mr \leqslant [\langle \_ \twoheadrightarrow mr, d, \mathsf{mR}, \overline{e_1}\rangle, \langle mr \rightarrow \_, \_, \mathsf{ivM}, \_\rangle^a, \\
&\qquad\qquad \langle mr \twoheadleftarrow, \_, \_\rangle^a, \langle mr \rightarrow \_, \_, \mathsf{ivR}, \_\rangle^b, \langle mr \twoheadleftarrow, \_, \_\rangle^b, \\
&\qquad\qquad\qquad \langle \leftarrow mr, d, \mathsf{mR}, e_1\rangle \,.\, \mathbf{some}\ d]^*
\end{aligned}
$$

The global invariant of a system S with the objects, $w_1 : Worker$, $w_2 : Worker$, $w_3 : Worker$, $mr : MapReduce(wp)$ and $m : Main(mr)$ is then

$$
I_S(h) \triangleq wf(h) \wedge I_{m:Main(mr)}(h) \wedge I_{mr:MapReduce(wp)}(h) \bigwedge_{i\in\{1,2,3\}} I_{w_i:Worker}(h)
$$

where well-formedness allows us to relate the different object histories. From this global invariant we may derive that the Reduce phase will starts only after the Map phase has been completed. Besides, none of the requests sent out to the workers is uncompleted when the call to mR on mr is finished.

As a special case, we consider a system where the instance of $Main$ invokes mR only once, i.e. $I_{m:Main(mr)}(h) \triangleq h/m \leqslant [\langle m \rightarrow mr, u, \mathsf{mR}, \overline{e}\rangle \,.\, \mathbf{some}\ u]$. History well-formedness then ensures that the cycles defined by the remaining invariants are repeated at most once, and that variables in the patterns are connected, i.e., the future u in $I_{m:Main(mr)}$ is identical to the future d in $I_{mr:MapReduce(wp)}$. The global invariant then reduces to the following:

$$
\begin{aligned}
I_S(h) \triangleq\ & wf(h) \wedge h/m \leqslant [\langle m \rightarrow mr, u, \mathsf{mR}, \overline{e}\rangle] \\
& \wedge h/w_1 \leqslant [\langle mr \twoheadrightarrow w_1, u_1, \mathsf{ivM}, \overline{e_1}\rangle, \langle \leftarrow w_1, u_1, \mathsf{ivM}, e_1\rangle \mid \\
& \quad \langle mr \twoheadrightarrow w_1, u_2, \mathsf{ivR}, \overline{e_2}\rangle, \langle \leftarrow w_1, u_2, \mathsf{ivR}, e_2\rangle \,.\, \mathbf{some}\ u_1, u_2, \overline{e_1}, e_1, \overline{e_2}, e_2]^* \\
& \wedge h/w_2 \leqslant [\langle mr \twoheadrightarrow w_2, u_3, \mathsf{ivM}, \overline{e_3}\rangle, \langle \leftarrow w_2, u_3, \mathsf{ivM}, e_3\rangle \mid \\
& \quad \langle mr \twoheadrightarrow w_2, u_4, \mathsf{ivR}, \overline{e_4}\rangle, \langle \leftarrow w_2, u_4, \mathsf{ivR}, e_4\rangle \,.\, \mathbf{some}\ u_3, u_4, \overline{e_3}, e_3, \overline{e_4}, e_4]^* \\
& \wedge h/w_3 \leqslant [\langle mr \twoheadrightarrow w_3, u_5, \mathsf{ivM}, \overline{e_5}\rangle, \langle \leftarrow w_3, u_5, \mathsf{ivM}, e_5\rangle \mid \\
& \quad \langle mr \twoheadrightarrow w_3, u_6, \mathsf{ivR}, \overline{e_6}\rangle, \langle \leftarrow w_3, u_6, \mathsf{ivR}, e_6\rangle \,.\, \mathbf{some}\ u_5, u_6, \overline{e_5}, e_5, \overline{e_6}, e_6]^* \\
& \wedge h/mr \leqslant [\langle m \twoheadrightarrow mr, u, \mathsf{mR}, \overline{e}\rangle, \langle mr \rightarrow \_, \_, \mathsf{ivM}, \_\rangle^a, \\
& \quad \langle mr \twoheadleftarrow, \_, \_\rangle^a, \langle mr \rightarrow \_, \_, \mathsf{ivR}, \_\rangle^b, \langle mr \twoheadleftarrow, \_, \_\rangle^b, \\
& \quad \langle \leftarrow mr, u, \mathsf{mR}, e\rangle]
\end{aligned}
$$

This invariant allows a number of global histories, depending on the interleaving of the activities in the different objects. The history h presented first in this section satisfies the invariant, and represents one particular interleaving.

Based on the assumption of functional correctness of each class, we now can derive that the MapReduce object does output the correct number of occurrences of each word in the collection of documents.

6. Related work

Models for asynchronous communication without futures have been explored for process calculi with buffered channels [Hoa85], for agents with message-based communication [AFK$^+$93], for method-based communication [MBM08], and in particular for Java [FCO99]. Behavioral reasoning about distributed and object-oriented systems is challenging, due to the combination of concurrency, compositionality, and object orientation. Moreover, the gap in reasoning complexity between sequential and distributed, object-oriented systems makes tool-based verification difficult in practice. A survey of these challenges can be found in [AD12]. Soundness of the parallel composition rules for shared-variable concurrency and synchronous message passing are proved in [dRdBH$^+$01]. A Hoare Logic for concurrent processes (objects) is presented in [dB02]. The Hoare Logic is compositional, and soundness and relative completeness are proven. In contrast to our work, communication is by message passing rather than by futures, and the objects communicate through FIFO channels.

The present approach follows the line of work based on communication histories to model object communication events in a distributed setting [BS01, Dah77, Hoa85]. Objects are concurrent and interact solely by method calls and futures, and remote access to object fields are forbidden. By creating unique references for method calls, the *label* construct of Creol [JO07] resembles futures, as callers may postpone reading result values. Verification systems capturing Creol labels can be found in [AD12, DJO05]. However, a label reference is local to the caller, and cannot be shared with other objects. A reasoning system for futures has been presented in [dBCJ07], using a combination of global and local invariants. Futures are treated as visible objects rather than reflected by events in histories. In contrast to our work, global reasoning is obtained by means of global invariants, and not by compositional rules. Thus the environment of a class must be known at verification time. SCOOP [Mey93, MBM08] and Cameo [BP09] are two concurrency models for Eiffel [Mey97] based on the concepts of design-by-contract. Compared with our work, these two approaches are not using histories.

A reasoning system for asynchronous methods in ABS without futures is presented in [DDJO12]. We here define a five-event semantics reflected actions on shared futures and object creation. The semantics gives a clean separation of the activities of the different objects, which leads to disjointness of local histories. Thus, object behavior can be specified in terms of the observable interaction of the current object only. This is essential for obtaining a simple reasoning system. In related approaches, e.g., [AD12, DJO05], events are visible to more than one object. The local histories must then be updated with the activity of other objects, resulting in more complex reasoning systems. Based on the five-event semantics, we present a compositional reasoning system for distributed, concurrent objects with asynchronous method calls. A class invariant defines a relation between the inner state and the observable communication of instances, and can be verified independently for each class. The class invariant can be instantiated for each object of the class, resulting in a history invariant over the observable behavior of the object. Compositional reasoning is ensured as history invariants may be combined to form global system specifications. The composition rule is similar to [DDJO12], which is inspired by previous approaches [Sou84a, Sou84b]. This work is an extension of our former paper [DDO12b]. Here we analyze a larger case study using futures, extend the language and semantics, including object creation, branching and looping constructs. Also, soundness proofs for class reasoning and in particular object composition are provided.

7. Conclusion

In this paper we have considered concurrent objects communicating by means of futures and a notion of non-blocking methods calls. This concurrency model is different from that found in mainstream languages such as Java. We find it interesting since it is based on high-level synchronization primitives, rather than locks and signaling, and it allows the caller to control the waiting time by means of different ways of calling a method, suspending, blocking, or non-blocking. In addition it directly supports distribution, autonomy, message-based communication, and object-orientation. Thus the class mechanism is devoted to programming of concurrent and autonomous objects, whereas internal data structures are programmed by the use of data types. This concurrency model has recently been the theme of several EU projects, including Credo, Hats, Envisage, and Upscale. Tool support for this concurrency model have been investigated in several ways, including compilation of the ABS language to more low-level languages including Java, Erlang, and Scala. The concurrency model gives rise to a clean, compositional semantics. Compositionality is a key property for scalability, allowing program units to be developed, tested, and understood, independently. The concurrency model may be relevant for Java, extended with asynchronous methods, and restricted so that remote access, notification, and explicit locking, are avoided.

The focus of this paper is program reasoning, and the concurrency model is chosen due to advantages with respect to program reasoning, while supporting true concurrency of objects. Compositional reasoning is facilitated by expressing object properties in terms of observable interaction between the object and its environment, recorded on communication histories. Object generation is reflected in the history by means of object creation events. A method call cycle with multiple future readings is reflected by four kinds of events, giving rise to disjoint communication alphabets for different objects. Specifications in terms of history invariants may then be derived independently for each object and composed in order to derive properties for concurrent object systems. At the class level, invariants define relationships between class attributes and the observable communication of class instances. The presented reasoning system is proved sound with respect to the given operational semantics. This system is easy to apply in the sense that class reasoning is similar to standard sequential reasoning, but with the addition of effects on the local history for statements involving futures. In particular, reasoning inside classes is not affected by the complexity of concurrency and synchronization; and one may express assumptions about inputs from the environment when convenient.

The main result of this paper is soundness of the rule for composition objects running in parallel. A minor result is that sound composition requires the query rule to have a condition related to the invariant, not found in earlier papers. We consider here global history invariants that are continuously satisfied, in the sense that any reachable global configuration of an object system must satisfy the invariant. The condition on the query rule would not be needed with a weaker notion of global history invariants stating that the global invariant holds as long as all objects are live (not blocked). Verification-wise the condition on the query rule, is somewhat similar to a query statement releasing the processor, as for instance the **await** *future* statement of the ABS language. Semantically, a blocking query has the advantage that it does not change the state, whereas a non-blocking query gives a state satisfying the local invariant. Thus the combination of the query and processor release mechanisms will not add significant verification complexity, and is also attractive from a programming perspective.

In order to focus on the future mechanism, this paper considers a core language with shared futures. The report version [DDO12a] considers a richer language, including constructs for inter-object process control and processor release. The verification system is suitable for an implementation within the KeY framework. With support for (semi-)automatic verification, such an implementation will be valuable when developing larger case studies. It is also natural to investigate how our reasoning system would benefit from extending it with rely/guarantee style reasoning [dRdBH$^+$01]. Assumptions about callee behavior may, for instance, be used to express properties of return values. More sophisticated techniques may also be used, e.g., [DO98, JO04] adapts rely/guarantee style reasoning to history invariants. However, such techniques requires more complex composition rules.

Acknowledgements

This work is supported by the EU Project FP7-610582 *Envisage: Engineering Virtualized Services* (http://www.envisage-project.eu) and FP7-ICT-2013-X *UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations* (http://www.upscale-project.eu).

References

[AD12] Ahrendt W, Dylla M (2012) A system for compositional verification of asynchronous objects. Sci Comput Program. 77(12):1289-1309. doi:10.1016/j.scico.2010.08.003

[AFK$^+$93] Agha G, Frølund S, Kim WY, Panwar R, Patterson A, Sturman D (1993) Abstraction and modularity mechanisms for concurrent computing. Parallel Distrib Technol Syst Appl IEEE 1(2):3–14

[ÁGGS09] Ábrahám E, Grabe I, Grüner A, Steffen M (2009) Behavioral interface description of an object-oriented language with futures and promises. J Log Algebr Program 78(7):491–518

[AS85] Alpern B, Schneider FB (1985) Defining liveness. Inf Process Lett 21(4):181–185

[AY07] Ahern A, Yoshida N (2007) Formalising java rmi with explicit code mobility. Theor Comput Sci 389(3):341–410. Semantic and Logical Foundations of Global Computing

[BHS07] Beckert B, Hähnle R, Schmitt PH (eds) (2007) Verification of object-oriented software: the KeY approach. LNCS, vol 4334. Springer, Berlin

[BJH77] Baker Jr HG, Hewitt C (1977) The incremental garbage collection of processes. In: Proceedings of the 1977 symposium on artificial intelligence and programming languages, New York, NY, USA. ACM, pp 55–59

[BP09] Brooke PJ, Paige RF (2009) Cameo: an alternative model of concurrency for Eiffel. Form Asp Comput 21(4):363–391

[BS01] Broy M, Stølen K (2001) Specification and development of interactive systems. Monographs in computer science. Springer, Berlin

[CDE$^+$07] Clavel M, Durán F, Eker S, Lincoln P, Martí-Oliet N, Meseguer J, Talcott CL (2007) All about Maude—a high-performance logical framework, how to specify, program and verify systems in rewriting logic. LNCS, vol 4350. Springer, Berlin

[Dah77] Dahl O-J (1977) Can program proving be made practical? In: Amirchahy M, Néel D (eds) Les Fondements de la Programmation. Institut de Recherche d'Informatique et d'Automatique, Toulouse, France, December 1977, pp 57–114

[Dah87] Dahl O-J (1987) Object-oriented specifications. In: Research directions in object-oriented programming. MIT Press, Cambridge, pp 561–576

[Dah92] Dahl O-J (1992) Verifiable programming. International series in computer science. Prentice Hall, New York

[dB02] de Boer FS (2002) A Hoare logic for dynamic networks of asynchronously communicating deterministic processes. Theor Comput Sci 274:3–41

[dBCJ07] de Boer FS, Clarke D, Johnsen EB (2007) A complete guide to the future. In: de Nicola R (ed) Proceedings of the 16th European symposium on programming (ESOP'07), March 2007. LNCS, vol 4421. Springer, Berlin, pp 316–330

[DDJO12] Din CC, Dovland J, Johnsen EB, Owe O (2012) Observable behavior of distributed systems: component reasoning for concurrent objects. J Log Algebr Program 81(3):227–256

[DDO12a] Din CC, Dovland J, Owe O (2012) An approach to compositional reasoning about concurrent objects and futures. Research Report 415, Department of Informatics, University of Oslo, February 2012. http://urn.nb.no/URN:NBN:no-30589

[DDO12b] Din CC, Dovland J, Owe O (2012) Compositional reasoning about shared futures. In: Eleftherakis G, Hinchey M, Holcombe M (eds) Proceedings of the international conference on software engineering and formal methods (SEFM'12). LNCS, vol 7504. Springer, Berlin, pp 94–108

[DG08] Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Commun. ACM 51(1):107–113

[DJO05] Dovland J, Johnsen EB, Owe O (2005) Verification of concurrent objects with asynchronous method calls. In: Proceedings of the IEEE international conference on software science, technology and engineering (SwSTE'05), February 2005. IEEE Computer Society Press, pp 141–150

[DO98] Dahl O-J, Owe O (1998) Formal methods and the RM-ODP. Research Report 261, Department of Informatics, University of Oslo, Norway, May 1998

[dRdBH$^+$01] de Roever W-P, de Boer F, Hannemann U, Hooman J, Lakhnech Y, Poel M, Zwiers J (2001) Concurrency verification: introduction to compositional and noncompositional methods. Cambridge University Press, New York

[FCO99] Falkner KEK, Coddington PD, Oudshoorn MJ (1999) Implementing asynchronous remote method invocation in java

[HAT] Full ABS Modeling Framework (2011). Deliverable 1.2 of project FP7-231620 (HATS). http://www.hats-project.eu

[HJ85] Halstead Jr RH (1985) Multilisp: a language for concurrent symbolic computation. ACM Trans Program Lang Syst 7(4):501–538

[Hoa85] Hoare CAR (1985) Communicating sequential processes. International series in computer science. Prentice Hall, Englewood Cliffs

[Int95] International Telecommunication Union (1995) Open distributed processing-reference model parts 1–4. Technical report, ISO/IEC, Geneva, July 1995

[JO04] Johnsen EB, Owe O (2004) Object-oriented specification and open distributed systems. In: Owe O, Krogdahl S, Lyche T (eds) From object-orientation to formal methods: essays in memory of Ole-Johan Dahl. LNCS, vol 2635. Springer, Berlin, pp 137–164

[JO07] Johnsen EB, Owe O (2007) An asynchronous communication model for distributed concurrent objects. Softw Syst Model 6(1):35–58

[JR05] Jeffrey ASA, Rathke J (2005) Java Jr.: fully abstract trace semantics for a core Java language. In: Proceedings of the European symposium on programming. LNCS, vol 3444. Springer, Berlin, pp 423–438

[LS88] Liskov BH, Shrira L (1988) Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In: Wise DS (ed) Proceedings of the SIGPLAN conference on programming language design and implementation (PLDI'88). ACM Press, pp 260–267

[MBM08] Morandi B, Bauer SS, Meyer B (2008) SCOOP—a contract-based concurrent object-oriented programming model. In: Müller P (ed) Advanced lectures on software engineering, LASER Summer School 2007/2008. Lecture notes in computer science, vol 6029. Springer, Berlin, pp 41–90

[Mes92] Meseguer J (1992) Conditional rewriting logic as a unified model of concurrency. Theor Comput Sci 96:73–155

[Mey93] Meyer B (1993) Systematic concurrent object-oriented programming. Commun. ACM 36(9):56–80

[Mey97] Meyer B (1997) Object-oriented software construction, 2nd edn. Prentice-Hall, Inc., Upper Saddle River

[Sou84a] Soundararajan N (1984) Axiomatic semantics of communicating sequential processes. ACM Trans Program Lang Syst 6(4):647–662

[Sou84b] Soundararajan N (1984) A proof technique for parallel programs. Theor Comput Sci 31(1–2):13–29

[YBS86] Yonezawa A, Briot J-P, Shibayama E (1986) Object-oriented concurrent programming in ABCL/1. In: Conference on object-oriented programming systems, languages and applications (OOPSLA'86). Sigplan Notices, vol 21, no 11, pp 258–268, November 1986

Received 10 September 2013

Revised 5 September 2014

Accepted 1 October 2014 by George Eleftherakis, Mike Hinchey, and Michael Butler

Appendix B

The Deductive Verification Tool: KeY-ABS

KeY-ABS: A Deductive Verification Tool for the Concurrent Modelling Language ABS[⋆]

Crystal Chang Din, Richard Bubel, and Reiner Hähnle

Department of Computer Science, Technische Universität Darmstadt, Germany
{crystald,bubel,haehnle}@cs.tu-darmstadt.de

Abstract. We present KeY-ABS, a tool for deductive verification of concurrent and distributed programs written in ABS. KeY-ABS allows to verify data dependent and history-based functional properties of ABS models. In this paper we give a glimpse of system workflow, tool architecture, and the usage of KeY-ABS. In addition, we briefly present the syntax, semantics and calculus of KeY-ABS Dynamic Logic (ABSDL). The system is available for download.

1 Introduction

KeY-ABS is a deductive verification system for the concurrent modelling language ABS [1,10]. It is based on the KeY theorem prover [2]. KeY-ABS provides an interactive theorem proving environment and allows one to prove properties of object-oriented and concurrent ABS models. The concurrency model of ABS has been carefully engineered to admit a proof system that is modular and permits to reduce correctness of concurrent programs to reasoning about sequential ones [3, 5]. The deductive component of KeY-ABS is an axiomatization of the operational semantics of ABS in the form of a sequent calculus for first-order dynamic logic for ABS (ABSDL) . The rules of the calculus that axiomatize program formulae define a symbolic execution engine for ABS. The system provides heuristics and proof strategies that automate large parts of proof construction. For example, first-order reasoning, arithmetic simplification, symbolic state simplification, and symbolic execution of loop- and recursive-free programs are performed mostly automatically. The remaining user input typically consists of universal and existential quantifier instantiations.

ABS is a rich language with Haskell-like (first-order) datatypes, Java-like objects and thread-based as well as actor-based concurrency. In contrast to model checking, KeY-ABS allows to verify complex functional properties of systems with unbounded size [6]. In this paper we concentrate on the design of the KeY-ABS prover and its usage. KeY-ABS itself consists of around 11,000 lines of Java code (KeY-ABS + reused parts of KeY: ca. 100,000 lines in total). The rule base consists of around 10,000 lines written in KeY's *taclet* rule description language [2]. At http://www.envisage-project.eu/?page_id=1558 the KeY-ABS tool can be downloaded.

[⋆] This work was done in the context of the EU project FP7-610582 *Envisage: Engineering Virtualized Services* (http://www.envisage-project.eu)

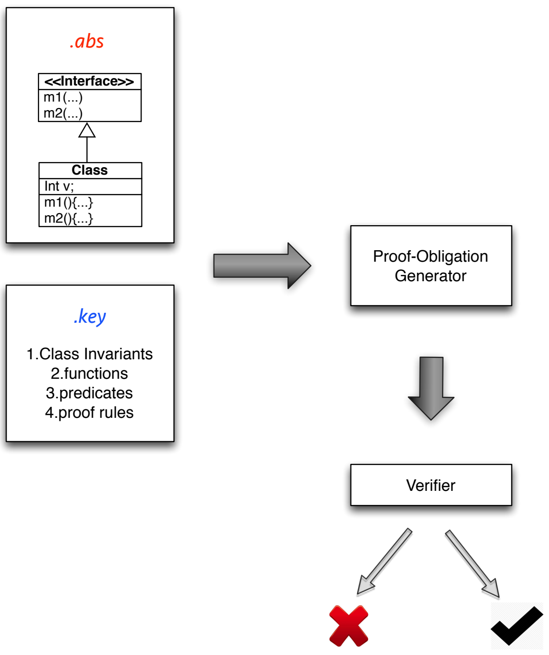


Fig. 1: Verification workflow of KeY-ABS

2 The Design of KeY-ABS

2.1 System Workflow

The input files to KeY-ABS comprise (i) an *.abs* file containing ABS programs and (ii) a *.key* file containing the class invariants, functions, predicates and specific proof rules required for this particular verification case. Given these input files, KeY-ABS opens a proof obligation selection dialogue that lets one choose a target method implementation. From the selection the proof obligation generator creates an ABSDL formula. By clicking on the **Start** button the verifier will try to automatically prove the generated formula. A positive outcome shows that the target method preserves the specified class invariants. In the case that a subgoal cannot be proved automatically, the user is able to interact with the verifier to choose proof strategies and proof rules manually. The reason for a formula to be unprovable might very well be that the target method implementation does not preserve one of the class invariants, that the specified invariants are too weak/too strong or that additional proof rules are required. The workflow of KeY-ABS is illustrated in Fig. 1.

2.2 The Concurrency Model of ABS

In ABS [1, 10] two different kinds of concurrency are supported depending on whether two objects belong to the same or to different *concurrent object groups* (COGs). The affinity of an object to a COG is determined at creation time. The creator decides whether the object should be assigned to a new COG or to the COG of its creator. Within a COG several threads might exist, but only one of these threads (and hence one object) can be active at any time. Another thread can only take over when the current active thread explicitly releases control. In other words, ABS realizes *cooperative scheduling* within a COG. All interleaving points occur syntactically explicit in an ABS program in the form of an *await* or *suspend* statement by which the current thread releases control.

While one COG represents a single processor with task switching and shared memory, two different COGs run actually in parallel and are separated by a network. As a consequence, objects within the same COG may communicate either by asynchronous or by synchronous method invocation, while objects living on different COGs *must* communicate with asynchronous method invocation and message passing. Any asynchronous method invocation creates a so called *future* as its immediate result. Futures are a handle for the result value once it becomes available. Attempting to access an unavailable result blocks the current thread and its COG until the result value is available. To avoid this, retrieval of futures is usually guarded with an *await* that, instead of blocking, releases control in case of an unavailable result. Futures are first-class citizens and can be assigned to local variables, object fields, and passed as method arguments.

2.3 Verification Approach for ABS Programs

Object fields are private and may only be accessed by the object they belong to (this is a stronger notion of privacy than in Java where other instances of the same class can access private fields). Hence, aliasing does not pose any problem for the verification of ABS.

To make verification of ABS programs modular, we KeY-ABS follows the monitor [8] approach. We define invariants for each ABS class to reason locally about classes. Each class invariant is required to hold after initialization in all class instances, before any process release point, and upon termination of each method call. Consequently, whenever a process is released, either by termination of a method execution or by a release point, the thread that gains execution control can rely on the class invariant to hold.

To write meaningful invariants of concurrent systems, it must be possible to refer to previous communication events. The observable behavior of a system can be described by *communication histories* over observable events [9]. Since message passing in ABS is *asynchronous*, in KeY-ABS we consider separate events for method invocation, for reacting upon a method call, for resolving a future, and for fetching the value of a future. Each event can only be observed by one object, namely the object that generates it. Assume an object o calls a method on object o' and generates a future identity fr associated to the method call. An invocation

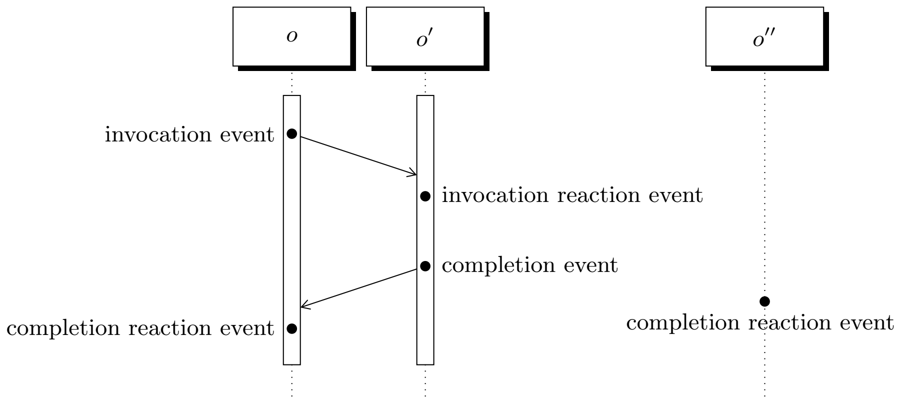

o
o'
o''
invocation event
invocation reaction event
completion event
completion reaction event
completion reaction event

Fig. 2: History events and when they occur on objects o, o' and o''

message is sent from o to o' when the method is invoked. This is reflected by the *invocation event* generated by o and illustrated by the sequence diagram in Fig. 2. An *invocation reaction event* is generated by o' once the method starts execution. When the method terminates, the object o' generates the *completion event*. This event reflects that the associated future is resolved, i.e., it contains the called method's result. The *completion reaction event* is generated by the caller o when fetching the value of the resolved future. Since future identities may be passed to a third object o'', that object may also fetch the future value, reflected by another *completion reaction event*, generated by o'' in Fig. 2.

2.4 Syntax and Semantics of the KeY-ABS Logic

Specification and verification of ABS models is done in ABS dynamic logic (ABSDL). ABSDL is a typed first-order logic plus a box modality: For an ABS program S and ABSDL formulae P and Q, the formula $P \rightarrow [\mathsf{S}]Q$ expresses: If the execution of a program S starts in a state where the assertion P holds and the program terminates normally, then the assertion Q holds in the final state. Hence, $[\cdot]$ acts as a partial correctness modality operator. Verification of an ABSDL formula proceeds by symbolic execution of S, where state modifications are handled by the *update* mechanism [2]. An *elementary update* has the form $U = \{loc := val\}$, where loc is a *location* expression and val is its new value term. Updates can only be applied to formulae or terms. Semantically, the validity of $U\phi$ in state s is defined as the validity of ϕ in state s', which is state s where the values of loc are modified according to update U. There are operations for sequential as well as parallel composition of updates. Typically, loop- and recursion-free sequences of program statements can be turned into updates fully automatical. Given an ABS method m with body mb and a class invariant I, the ABSDL formula $I \rightarrow [mb]I$ expresses that the method m preserves the class invariant.

In ABSDL we express properties of a system in terms of histories. This is realized by a dedicated, global program variable history, which contains the object local histories as a sequence of events. The history events themselves are elements of datatype HistoryLabel, which defines for each event type a constructor function. For instance, a completion event is represented as $compEv(o, fr, m, e)$ where o is the callee, fr the corresponding future, m the method name, and e the return result of the method execution. In addition to the history formalisation as a sequence of events, there are a number of built-in functions and predicates that allow to express common properties concerning histories. For example, function *getFuture*(e) returns the future identity contained in the event e, and predicate *isInvocationEv*(e) returns true if event e is an invocation event.

The type system of KeY-ABS reflects the ABS type system. Besides HistoryLabel, the type system of ABSDL contains, for example, the sequence type Seq, the root reference type any, the super type ABSAnyInterface of all ABS objects, the future type Future, and the type null, which is a subtype of all reference types. Users can define their own functions, predicates and types, which are used to represent the interfaces and abstract data types of a given ABS program.

2.5 Rule Formalisation

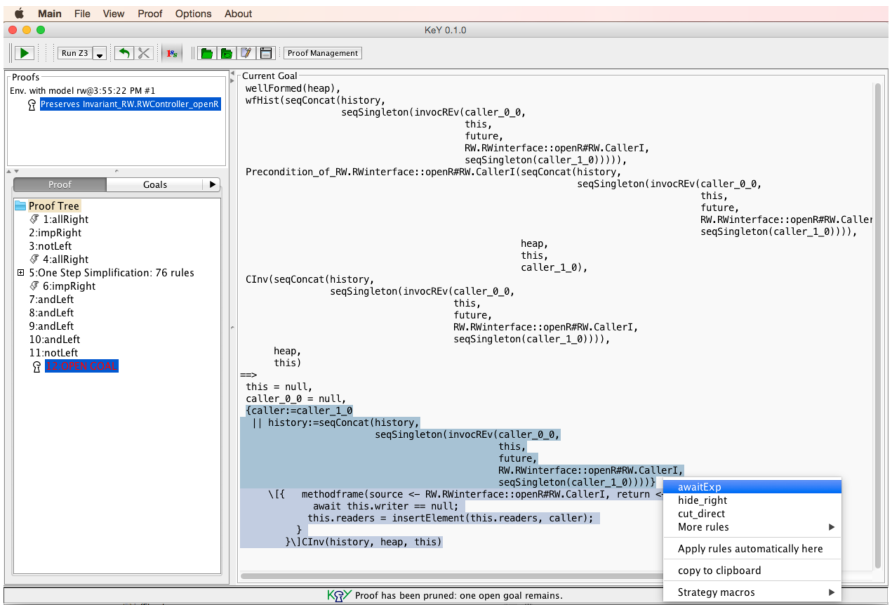
The user can interleave the automated proof search implemented in KeY-ABS with interactive rule application. For the latter, the KeY-ABS prover has a graphical user interface that is built upon the idea of direct manipulation. To apply a rule, the user first selects a focus of application by highlighting a (sub-)formula or a (sub-)term in the goal sequent. The prover then offers a choice of rules applicable at this focus. Rule schema variable instantiations are mostly inferred by matching. Fig. 3 shows an example of proof rule selection in KeY-ABS. The user is about to apply the *awaitExp* rule that executes an await statement.

Another way to apply rules and provide instantiations is by drag and drop. The user simply drags an equation onto a term, and the system will try to rewrite the term with the equation. If the user drags a term onto a quantifier the system will try to instantiate the quantifier with this term.

The interaction style is closely related to the way rules are formalised in KeY-ABS. All rules are defined as *taclets* [2]. Here is a (slightly simplified) example:

\find $([\{method(source \leftarrow m, return \leftarrow (var : r, fut : u)) : \{\mathsf{return}\ exp; \}\}]\phi)$
\replacewith $(\{\mathsf{history} := seqConcat(\mathsf{history}, compEv(this, u, m, exp))\}\phi)$
\heuristics $(simplify\_prog)$

The rule symbolically executes a return statement inside a method invocation. It applies the *update* mechanism to the variable history, which is extended with a *completion event* capturing the termination and return value of the method execution. The **find** clause specifies the potential application focus. The taclet will be offered to the user on selecting a matching focus. The action clause **replacewith** modifies the formula in focus. The **heuristics** clause provides priority information to the parameterized automated proof search strategy. The



```
Env. with model rw@3:55:22 PM #1
  Preserves Invariant_RW.RWController_openR
```

```
Proof Tree
   1:allRight
  2:impRight
  3:notLeft
   4:allRight
⊞ 5:One Step Simplification: 76 rules
   6:impRight
  7:andLeft
  8:andLeft
  9:andLeft
  10:andLeft
  11:notLeft
   12:OPEN GOAL
```

```
 wellFormed(heap),
 wfHist(seqConcat(history,
                  seqSingleton(invocREv(caller_0_0,
                                        this,
                                        future,
                                        RW.RWinterface::openR#RW.CallerI,
                                        seqSingleton(caller_1_0))))),
 Precondition_of_RW.RWinterface::openR#RW.CallerI(seqConcat(history,
                                                            seqSingleton(invocREv(caller_0_0,
                                                                                  this,
                                                                                  future,
                                                                                  RW.RWinterface::openR#RW.Caller
                                                                                  seqSingleton(caller_1_0)))),
                                                 heap,
                                                 this,
                                                 caller_1_0),
 CInv(seqConcat(history,
                seqSingleton(invocREv(caller_0_0,
                                      this,
                                      future,
                                      RW.RWinterface::openR#RW.CallerI,
                                      seqSingleton(caller_1_0)))),
      heap,
      this)
==>
 this = null,
 caller_0_0 = null,
 {caller:=caller_1_0
  || history:=seqConcat(history,
                        seqSingleton(invocREv(caller_0_0,
                                              this,
                                              future,
                                              RW.RWinterface::openR#RW.CallerI,
                                              seqSingleton(caller_1_0))))}
    \[{   methodframe(source <- RW.RWinterface::openR#RW.CallerI, return <
            await this.writer == null;
           this.readers = insertElement(this.readers, caller);
          }
        }\]CInv(history, heap, this)
```

Proof has been pruned: one open goal remains.

Fig. 3: Proof rule selection

taclet language is quickly mastered and makes the rule base easy to maintain and extend. A full account of the taclet language is given in [2].

2.6 KeY-ABS Architecture

Fig. 4 depicts the principle architecture of the KeY-ABS system. KeY-ABS is based on the KeY 2.0 platform—a verification system for Java. To be able to reuse most parts of the system, we had to generalize various subsystems and to abstract away from their Java specifics. For instance, the rule application logic of KeY made several assumptions which are valid for Java but not for other programming languages. Likewise, the specification framework of KeY, even though it provided general interfaces for contracts and invariants, made implicit assumptions that were insufficient for our communication histories and needed to be factored out. After refactoring the KeY system provides core subsystems (rule engine, proof construction, search strategies, specification language, proof management etc.) that are inde-

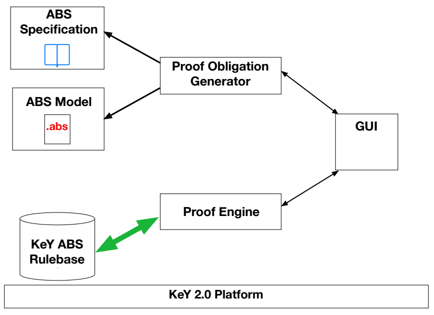


Fig. 4: The architecture of KeY-ABS

pendent of the specific program logic or target language. These are then extended and adapted by the ABS and Java backends.

The proof obligation generator needs to parse the source code of the ABS model and the specification. For the source code we use the parser as provided by the ABS toolkit [13] with no changes. The resulting abstract syntax tree is then converted into KeY's internal representation. The specification parser is an extension of the parser for ABSDL logic formulas and is an adapted version of the parser for JavaDL [2]. The rule base for ABSDL reuses the language-independent theories of the KeY tool, such as arithmetic, sequences and first-order logic. The rules for symbolic execution have been written from scratch for ABS as well as the formalisation of the history datatype.

3 The Usage of KeY-ABS

The ABS language was designed around a concurrency model whose analysis stays manageable. The limitations of the ABS concurrency model, specifically the fact that scheduling points are syntactically explicit, makes it possible to define a compositional specification and verification method. This is essential for being able to scale verification to non-trivial programs, because it is possible to specify and verify each ABS method separately, without the need for a global invariant. KeY-ABS follows the Design-by-Contract paradigm with an emphasis on specification of class invariants for ABS programs.

```
class RWController implements RWinterface {
  Set<CallerI> readers = EmptySet; CallerI writer = null;

  Unit openR(CallerI caller){
    await writer == null;
    readers = insertElement(readers, caller);}

  Unit closeR(CallerI caller){
    readers = remove(readers, caller);}

  Unit openW(CallerI caller){
    await writer == null;
    writer = caller; readers = insertElement(readers, caller);}

  Unit closeW(CallerI caller){
    await writer == caller;
    writer = null; readers = remove(readers, caller);}

  String read(CallerI caller, Int key){...}

  Unit write(CallerI caller, Int key, String value){...}
}
```

Fig. 5: The controller class of the RW example in ABS

A history-based class invariant in ABSDL can relate the state of an object to the local history of the system. A simple banking system is verified in [3] by

KeY-ABS, where an invariant ensures that the value of the account balance (a class attribute) always coincides with the value returned by the most recent call to a deposit or withdraw method (captured in the history). Here we use a more ambitious case study to illustrate this style of class invariant. In Fig. 5 an ABS implementation of the classic reader-writer problem [4] is shown. The RWController class provides read and write operations to clients and four methods to synchronize reading and writing activities: openR, closeR, openW and closeW.

The class attribute *readers* contains a set of clients currently with read access and *writer* contains the client with write access. The set of *readers* is extended by execution of openR or openW, and is reduced by closeR or closeW. The *writer* is added by execution of openW and removed by closeW. Two class invariants of the reader-writer example are (slightly simplified) shown in Fig. 6, in which the invariants $isReader$ and $isWriter$ express that the value of class attributes *readers* and *writer* are always equal to the set of relevant callers extracted from the current history. The keyword **invariants** opens a section where invariants can be specified. Its parameters declare program variables that can be used to refer to the history (historySV), the heap (heapSV, implicit by attribute access), and the current object (self, similar as Java's *this*).

The functions $currentReaders(h)$ and $currentWriter(h)$ are defined inductively over the history h to capture a set of existing callers to the corresponding methods. The statistics of verifying these two invariants are in Fig. 7. For each of the six methods of the RWController class we show it satisfies $isReader$ and $isWriter$. For instance, the proof of invariant $isReader$ requires 3884 proof steps for method openR and the corresponding proof tree contains 12 branches. Verification of this case study was automatic except for a few instantiations of quantifiers and the rule application on inductive functions.[1]

```
\invariants(Seq historySV, Heap heapSV, ABSAnyInterface self) {
  isReader : RW.RWController {
    RW.RWController::self.readers
    = currentReaders(historySV)
  };

  isWriter : RW.RWController {
    insertElement(EmptySet, RW.CallerI::self.writer)
    = currentWriter(historySV)
  };
}
```

Fig. 6: Class invariants of the RW example

[1] The complete model of the reader-writer example with all formal specifications and proofs is available at
https://www.se.tu-darmstadt.de/se/group-members/crystal-chang-din/rw.

invariants\methods	openR	closeR	openW	closeW	read	write
isReader	3884 – 12	1147 – 7	2836 – 9	3904 – 12	5459 – 26	3572 – 35
isWriter	2735 – 9	739 – 5	3891 – 12	3818 – 12	5056 – 29	4058 – 32

Fig. 7: Verification Result of RW example: # nodes – # branches

A history-based class invariant in ABSDL can also express temporal or structural properties of the history, for example, that an event ev_1 occurs in the history before an event ev_2 is generated. To formalize this kind of class invariant, quantifiers and indices of sequences are used to locate the events at certain positions of the history. Recently, we applied the KeY-ABS system to a case study of an ABS model of a Network-on-Chip (NoC) packet switching platform [11], called ASPIN (Asynchronous Scalable Packet Switching Integrated Network) [12]. We proved that ASPIN drops no packets. The ABS model, the specifications and the proof rules can be found in [6]. Both styles of class invariants mentioned above were used. The KeY-ABS verification approach to the NoC case study deals with an *unbounded* number of objects and is valid for generic NoC models for any m $\times$ n mesh in the ASPIN chip as well as any number of sent packets.

The global history of the whole system is formed by the composition of the local history of each instance of the class. A *global invariant* can be obtained as a conjunction of the class invariants verified by KeY-ABS for all objects in the system, adding wellformedness of the global history [7]. This allows to prove *global* safety properties of the system using *local* rules and symbolic execution, such as absence of packet loss. In contrast to model checking this allows us to deal effectively with unbounded target systems without suffering from state explosion.

4 Conclusion

We presented the KeY-ABS formal verification tool for the concurrent modelling language ABS. ABS is a rich, fully executable language with unbounded data structures and Java-like control structures as well as objects. It offers thread-based as well as actor-based concurrency with the main limitation being that scheduling points are made syntactically explicit in the code ("cooperative scheduling"). KeY-ABS implements a compositional proof system [4, 5] for ABS. Its architecture is based on the state-of-art Java verification tool KeY and KeY-ABS reuses some of KeY's infrastructure.

KeY-ABS is able to verify global, functional properties of considerable complexity for unbounded systems. At the same time, the degree of automation is high. Therefore, KeY-ABS is a good alternative for the verification of unbounded, concurrent systems where model checking is not expressive or scalable enough. In the future we plan to use KeY-ABS for verification of concurrent algorithms with unbounded input.

References

1. *The ABS Language Specification*, 1.2.0 edition, Apr. 2013. http://tools.hats-project.eu/download/absrefmanual.pdf.
2. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.
3. R. Bubel, A. Flores Montoya, and R. Hähnle. Analysis of executable software models. In M. Bernardo, F. Damiani, R. Hähnle, E. B. Johnsen, and I. Schaefer, editors, *Executable Software Models: 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinoro, Italy*, volume 8483 of *LNCS*, pages 1–27. Springer, June 2014.
4. C. C. Din, J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. *Journal of Logic and Algebraic Programming*, 81(3):227–256, 2012.
5. C. C. Din and O. Owe. Compositional reasoning about active objects with shared futures. *Formal Aspects of Computing*, pages 1–22, 2014.
6. C. C. Din, S. L. T. Tarifa, R. Hähnle, and E. B. Johnsen. The NoC verification case study with KeY-ABS. Technical report, Department of Computer Science, Technische Universität Darmstadt, Germany, Feb. 2015.
7. J. Dovland, E. B. Johnsen, and O. Owe. Verification of concurrent objects with asynchronous method calls. In *Proc. IEEE Intl. Conference on Software Science, Technology & Engineering(SwSTE'05)*, pages 141–150. IEEE Computer Society Press, Feb. 2005.
8. C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.
9. C. A. R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice Hall, 1985.
10. 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 *Lecture Notes in Computer Science*, pages 142–164. Springer, 2011.
11. S. Kumar, A. Jantsch, M. Millberg, J. Öberg, J. Soininen, M. Forsell, K. Tiensyrjä, and A. Hemani. A network on chip architecture and design methodology. In *2002 IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2002), 25-26 April 2002, Pittsburgh, PA, USA*, pages 117–124, 2002.
12. A. Sheibanyrad, A. Greiner, and I. M. Panades. Multisynchronous and fully asynchronous NoCs for GALS architectures. *IEEE Design & Test of Computers*, 25(6):572–580, 2008.
13. 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. *STTT*, 14(5):567–588, 2012.

Appendix C

The NoC Verification Case Study with KeY-ABS

Formal Modeling, Specification, and Verification of Unbounded Network-on-Chips [⋆]

Crystal Chang Din[1], S. Lizeth Tapia Tarifa[2], Reiner Hähnle[1], and Einar Broch Johnsen[2]

[1] Department of Computer Science, Technische Universität Darmstadt, Germany
`{crystald,haehnle}@cs.tu-darmstadt.de`
[2] Department of Informatics, University of Oslo, Norway
`{sltarifa,einarj}@ifi.uio.no`

Abstract. We present a case study on scalable formal verification of distributed systems that involves a formal model of a Network-on-Chip (NoC) packet switching platform. We provide an executable model of a generic $m \times n$ mesh chip with unbounded number of packets, the formal specification of certain safety properties, and formal proofs that the model fulfills these properties. The modeling has been done in ABS, a language that is intended to permit scalable verification of detailed, precisely modeled, executable, concurrent systems. Our paper shows that this is indeed possible and so advances the state-of-art verification of NoC systems. It also demonstrates that deductive verification is a viable alternative to model checking for the verification of unbounded concurrent systems that can effectively deal with state explosion.

1 Introduction

This paper presents a case study on *scalable* formal verification of the behavior of distributed systems. We create a formal, executable model of a *Network-on-Chip* (NoC) [27] packet switching platform called ASPIN (Asynchronous Scalable Packet Switching Integrated Network) [32]. This is a practically relevant system whose correctness is of great importance for the network infrastructures where it is deployed.

We model the ASPIN router architecture in the formal, executable, concurrent modeling language ABS [21, 23]. We use ABS for a number of reasons: (i) it combines functional, imperative, and object-oriented programming styles, allowing intuitive, modular, high-level modeling of concepts, domain and data; (ii) ABS models are fully executable and model system behavior precisely [2]; (iii) ABS can model synchronous as well as asynchronous communication; (iv) ABS has been developed to permit scalable formal verification:

[⋆] Supported by the EU project FP7-610582 *Envisage: Engineering Virtualized Services* (`http://www.envisage-project.eu`) and FP7-612985 *UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations* (`http://www.upscale-project.eu`).

there is a program logic [18] as well as a compositional proof system [16] that permits to prove global system properties by reasoning about object-local invariants; (v) ABS comes with an IDE and a range of analysis as well as productivity tools [34], specifically, there is a formal verification tool called KeY-ABS [7].

The main contributions of this paper are as follows: (i) a formal model of a generic $m \times n$ mesh ASPIN chip in ABS with unbounded number of packets, as well as a packet routing algorithm; (ii) the formal specification of a number of safety properties which together ensure that no packets are lost; (iii) formal proofs, done with KeY-ABS, that the ABS model of ASPIN fulfills these safety properties.[3]

ABS has been developed with the explicit aim to permit scalable verification of detailed, precisely modeled, executable, concurrent systems. Our paper shows that this claim is indeed justified. In addition it advances the state-of-the-art with the first successful verification of a generic NoC model that has an unbounded number of nodes and packets. This has been achieved with manageable effort and thus shows that deductive verification is a viable alternative to model checking for the verification of concurrent systems that can effectively deal with state explosion.

The paper is organized as follows: Sect. 2 gives a brief introduction into the modeling language ABS, Sect. 3 details our formal specification approach to system behavior, Sect. 4 provides some formal background on deductive verification with expressive program logics, and Sect. 5 presents the ASPIN NoC case study. Sect. 6 explains how we achieved the formal specification and verification of the case study and gives details about the exact properties proved as well as the necessary effort. In Sect. 7 we sketch possible directions for future work and Sect. 8 discusses related work and concludes.

2 The ABS Modeling Language

ABS [21, 23] is a formal behavioral specification language with a Java-like syntax. It combines functional and imperative programming styles to develop abstract executable models. ABS targets the modeling of concurrent, distributed, and object-oriented systems. It has a formal syntax and semantics and has a clean integration of concurrency and object orientation based on concurrent object groups (COGs) [23, 29]. ABS permits synchronous as well as asynchronous communication [24] akin to Actors [1] and Erlang processes [4]. ABS offers a wide variety of complementary modeling alternatives in a concurrent and object-oriented framework that integrates algebraic datatypes, functional programming and imperative programming. Compared to object-oriented programming languages, ABS abstracts from low-level implementation choices such as imperative data structures, and compared to design-oriented languages like UML diagrams, it models data-sensitive control flow and it is executable.

[3] The complete model with all formal specifications and proofs is available at https://www.se.tu-darmstadt.de/se/group-members/crystal-chang-din/noc.

Syntactic categories.	*Definitions.*
T in GroundType	$T ::= B \mid D \mid I \mid D\langle\overline{T}\rangle$
A in Type	$A ::= N \mid T \mid N\langle\overline{A}\rangle$
x in Variable	$Dd ::= \textsf{\textbf{data}}\ D[\langle\overline{A}\rangle] = [\overline{Cons}];$
e in Expression	$Cons ::= Co[(\overline{A})]$
v in Value	$F ::= \textsf{\textbf{def}}\ A\ fn[\langle\overline{A}\rangle](\overline{A\ x}) = e;$
br in Branch	$e ::= x \mid v \mid Co[(\overline{e})] \mid fn(\overline{e}) \mid \textsf{\textbf{case}}\ e\ \{\overline{br}\}$
p in Pattern	$v ::= Co[(\overline{v})] \mid \textsf{\textbf{null}}$
	$br ::= p \Rightarrow e;$
	$p ::= \_ \mid x \mid v \mid Co[(\overline{p})]$

Fig. 1. Syntax for the functional layer of ABS. Terms $\overline{e}$ and $\overline{x}$ denote possibly empty lists over the corresponding syntactic categories, and square brackets [ ] optional elements.

In addition, ABS also provides explicit and implicit time-dependent behavior [6], the modeling of deployment variability [25] and the modeling of variability in software product line engineering [9]. However, these functionalities of the language are not used in this paper and will not be further discussed. The rest of this section focuses on the syntax of ABS which contains a functional layer and an imperative layer. The details of the sequential execution of several threads inside a COG is not used in the verification techniques showcased in this paper and therefore we focus on single-object COGs (i.e., concurrent objects).

2.1 The Functional Layer of ABS

The functional layer of ABS is used to model computations on the internal data of the imperative layer. It allows modelers to abstract from implementation details of imperative data structures at an early stage in the software design and thus allows data manipulation without committing to a low-level implementation choice. The functional layer combines a simple language for parametric algebraic data types (ADTs) and a pure first-order functional language. ABS includes a library with four predefined basic types (Bool, Int, String, and Unit), and parametric datatypes, (such as lists, sets, and maps). The predefined datatypes come with arithmetic and comparison operators, and the parametric datatypes have built-in standard functions. The type Unit is used as a return type for methods without explicit return value. All other types and functions are user-defined.

The formal syntax of the functional language is given in Fig. 1. The *ground types* T consist of basic types B as well as names D for datatypes and I for interfaces. In general, a type A may also contain type variables N (i.e., uninterpreted type names [28]). In *datatype declarations* Dd, a datatype D has a set of constructors $Cons$, each of which has a name Co and a list of types $\overline{A}$ for their arguments. *Function declarations* F have a return type A, a function name fn, a list of parameters $\overline{x}$ of types $\overline{A}$, and a function body e. Both datatypes and functions may be polymorphic and have a bracketed list of type parameters (e.g., Set<Bool>). The layered type system allows functions in the functional layer to be defined over types A which are parametrized by type variables but

Syntactic categories.	*Definitions.*
s in Stmt	$P ::= \overline{IF}\ \overline{CL}\ \{[\overline{T}\ \overline{x};]\ s\,\}$
e in Expr	$IF ::= \textbf{interface}\ I\ \{\,[\overline{Sg}]\,\}$
b in BoolExpr	$CL ::= \textbf{class}\ C\ [(\overline{T}\ \overline{x})]\ [\textbf{implements}\ \overline{I}]\ \{\,[\overline{T}\ \overline{x};]\ \overline{M}\}$
g in Guard	$Sg ::= T\ m\ ([\overline{T}\ \overline{x}])$
	$M ::= Sg\ \{[\overline{T}\ \overline{x};]\ s\,\}$
	$s ::= s; s \mid \textbf{skip} \mid x = rhs \mid \textbf{if}\ b\ \{\,s\,\}\ [\textbf{else}\ \{\,s\,\}] \mid \textbf{while}\ b\ \{\,s\,\}$
	$\mid \textbf{await}\ g \mid \textbf{suspend} \mid \textbf{return}\ e$
	$rhs ::= e \mid cm \mid \textbf{new}\ C\ (\overline{e})$
	$cm ::= e!m(\overline{e}) \mid x.\textbf{get}$
	$g ::= b \mid x? \mid g \wedge g$

Fig. 2. Syntax for the imperative layer of ABS.

only applied to ground types T in the imperative layer; e.g., the head of a list is defined for List<A> but applied to ground types such as List<Int>.

Expressions e include variables x, values v, constructor expressions $Co(\overline{e})$, function expressions $fn(\overline{e})$, and case expressions **case** e $\{\overline{br}\}$. *Values* v are expressions that have reached a normal form: constructors applied to values $Co(\overline{v})$ or **null**. *Case expressions* match a value against a list of branches $p \Rightarrow e$, where p is a pattern. Patterns are composed of the following elements: (1) wild cards _ which match anything, (2) variables x match anything if they are free or match against the existing value of x if they are bound, (3) values v which are compared literally, and (4) constructor patterns $Co(\overline{p})$ which match Co and then recursively match the elements $\overline{p}$. The branches are evaluated in the listed order, free variables in p are bound in the expression e.

2.2 The Imperative Layer of ABS

The imperative layer of ABS addresses concurrency, communication, and synchronization in the system design, and defines interfaces, classes, and methods in an object-oriented language with a Java-like syntax. In ABS, concurrent objects (single object COGs) are *active* in the sense that their run method, if defined, starts automatically upon creation.

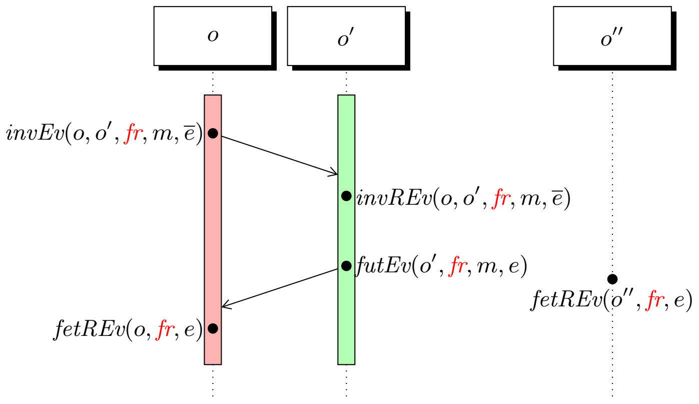
Statements are standard for sequential composition $s_1; s_2$, and for **skip**, **if**, **while**, and **return** constructs. Cooperative scheduling in ABS is achieved by explicitly suspending the execution of the active process. The statement **suspend** unconditionally suspends the execution of the active process and moves this process to the queue. The statement **await** g conditionally suspends execution: the guard g controls processor release and consists of Boolean conditions b and return tests $x?$ (explained in the next paragraph). Just like expressions e, the evaluation of guards g is side-effect free. However, if g evaluates to false, the processor is released and the process *suspended*. When the execution thread is idle, an enabled task may be selected from the pool of suspended tasks by means of a default scheduling policy. In addition to expressions e, the right hand side of an assignment $x{=}rhs$ includes object group creation **new** $C(\overline{e})$, method calls $o!m(\overline{e})$, and future dereferencing $x.$**get**. Method calls and future dereferencing are explained in the next paragraph.

Communication and *synchronization* are decoupled in ABS. Communication is based on asynchronous method calls, denoted by assignments of the form $f{=}o!m(\overline{e})$ to future variables f of type $\textbf{Fut}\langle T\rangle$, where T corresponds to the return type of the called method m. Here, o is an object expression, m a method name, and $\overline{e}$ are expressions providing actual parameter values for the method invocation. (Local calls are written $\textbf{this}!m(\overline{e})$.) After calling $f{=}o!m(\overline{e})$, the future variable f refers to the return value of the call, and the caller may proceed *without blocking*. Two operations on future variables control synchronization in ABS. First, the guard **await** f? *suspends the active process* unless a return to the call associated with f has arrived, allowing other processes in the object to execute. Second, the return value is retrieved by the expression f.**get**, which *blocks all execution* in the object until the return value is available. Futures are first-class citizens of ABS and can be passed around as method parameters. The read-only variable **destiny**() refers to the future associated with the current process [13]. The statement sequence $x{=}o!m(e)\,;v{=}x\,.\,\textbf{get}$ contains no suspension statement and, therefore, encodes commonly used *blocking calls*, abbreviated $v{=}o.m(e)$ (often referred to as synchronous calls). If the return value of a call is of no interest, the call may occur directly as a statement $o!m(e)$ with no associated future variable. This corresponds to asynchronous message passing.

The syntax of the imperative layer of ABS is given in Fig. 2. A program P consists of lists of interface and class declarations followed by a main block $\{\overline{T}\ \overline{x}; s\}$, which is similar to a method body. An interface IF has a name I and method signatures Sg. A class CL has a name C, interfaces $\overline{I}$ (specifying types for its instances), class parameters and state variables x of type T, and methods M (The *attributes* of the class are both its parameters and state variables). A method signature Sg declares the return type T of a method with name m and formal parameters $\overline{x}$ of types $\overline{T}$. M defines a method with signature Sg, local variable declarations $\overline{x}$ of types $\overline{T}$, and a statement s. Statements may access attributes, locally defined variables (including the read-only variables **this** for self-reference and **destiny**() explained above), and the method's formal parameters. There are no type variables at the imperative layer of ABS.

3 Observable Behavior

The observable behavior of a system can be described by *communication histories* over observable events [22]. Since message passing in ABS is *asynchronous*, we consider separate events for method invocation, reacting upon a method call, resolving a future, and for fetching the value of a future. Each event can only be observed by one object, namely the generating object. Assume an object o calls a method m on object o' with input values $\overline{e}$ and where fr denotes the identity of the associated future. An invocation message is sent from o to o' when the method is invoked. This is reflected by the *invocation event* $invEv(o, o', fr, m, \overline{e})$ generated by o and illustrated by the sequence diagram in Fig. 3. An *invocation reaction event* $invREv(o, o', fr, m, \overline{e})$ is generated by o' once the method starts execution. When the method terminates, the object o' generates the *fu-*



o
o'
o''
$invEv(o, o', fr, m, \overline{e})$
$invREv(o, o', fr, m, \overline{e})$
$futEv(o', fr, m, e)$
$fetREv(o'', fr, e)$
$fetREv(o, fr, e)$

Fig. 3. History events and when they occur

ture event $futEv(o', fr, m, e)$. This event reflects that fr is resolved with return value e. The *fetching event* $fetREv(o, fr, e)$ is generated by o when fetching the value of the resolved future. References fr to futures bind all four event types together and allow to filter out those events from an event history that relate to the same method invocation. Since future identities may be passed to another object o'', that object may also fetch the future value, reflected by the event $fetREv(o'', fr, e)$, generated by o'' in Fig. 3.

For a method call with future fr, the ordering of events is described by the regular expression

$$invEv(o, o', fr, m, \overline{e}) \cdot invREv(o, o', fr, m, \overline{e}) \cdot futEv(o', fr, m, e)[\cdot fetREv(\_, fr, e)]^*$$

for some fixed o, o', m, $\overline{e}$, e, and where "$\cdot$" denotes concatenation of events, "_" denotes arbitrary values. Thus the result value may be read several times, each time with the same value, namely that given in the preceding future event. A communication history is *wellformed* if the order of communication events follows the pattern defined above, the identities of the generated future is fresh, and the communicating objects are non-`null`.

Invariants Class invariants express a relation between the internal state and observable communication of class instances. They are specified by a predicate over the class attributes and the local history. A class invariant must hold after initialization, it must be maintained by all methods, and it must hold at all processor release points (i.e., `await`, `suspend`).

A *global invariant* can be obtained as a conjunction of the class invariants for all objects in the system, adding wellformedness of the global history [19]. This is made more precise in Sect. 6.2 below.

4 Deductive Verification

A formal proof is a sequence of reasoning steps designed to convince the reader about the truth of some formulae, i.e., a theorem. In order to do this the proof must lead without gaps from axioms to the theorem by applying proof rules.

KeY-ABS is a deductive verification system for ABS programs based on the KeY theorem prover [5]. As a program logic it uses first-order dynamic logic for ABS (ABSDL) [7, 16]. For an ABS program S and ABSDL formulae P and Q, the formula $P \to [S]Q$ expresses: If the execution of a program S starts in a state where the assertion P holds and the program terminates normally, then the assertion Q holds in the final state. Hence, $[\cdot]$ acts as a partial correctness modality operator. Given an ABS method m with body mb and a class invariant I, the ABSDL formula $I \to [mb]I$ expresses that the method m preserves the class invariant. We use a Gentzen-style sequent calculus to prove ABSDL formulae. Within a sequent we represent $P \to [S]Q$ as

$$\Gamma, P \vdash [S]Q, \Delta,$$

where Γ and Δ stand for (possibly empty) sets of formulae. A sequent calculus as realized in ABSDL essentially simulates a symbolic interpreter for ABS. The assignment rule for a local program variable is :

$$\frac{\Gamma \vdash \{\mathtt{v} := \mathtt{e}\}[\mathtt{rest}]\phi, \Delta}{\Gamma \vdash [\mathtt{v} = \mathtt{e}; \mathtt{rest}]\phi, \Delta}$$

where `v` is a local program variable and `e` is a pure (side effect-free) expression. This rule rewrites the formula by moving the assignment from the program into a so-called update $\{\mathtt{v} := \mathtt{e}\}$, which captures state changes. The symbolic execution continues with the remaining program `rest`. Updates [5] can be viewed as explicit substitutions that accumulate in front of the modality during symbolic program execution. Updates can only be applied to formulae or terms. Once the program to be verified has been completely executed and the modality is empty, the accumulated updates are applied to the formula after the modality, resulting in a pure first-order formula. Below we show the proof rule for asynchronous method invocations:

$$\mathsf{asyncCall}\ \frac{\begin{array}{l}\Gamma \vdash \{\mathcal{U}\}(o \not\doteq \mathtt{null} \land \mathtt{wf}(h)), \Delta \\ \Gamma \vdash \{\mathcal{U}\}(\mathtt{futureIsFresh}(u, h) \to \\ \qquad \{\mathtt{fr} := u \,||\, h := h \cdot \mathit{invEv}(\mathit{this}, o, u, m, \overline{e})\}[\mathtt{rest}]\phi), \Delta\end{array}}{\Gamma \vdash \{\mathcal{U}\}[\mathtt{fr} = o\mathtt{!m}(\overline{e}); \mathtt{rest}]\phi, \Delta}$$

This proof rule has two premisses and splits the proof into two branches. The first premiss on top ensures that the callee is non-null and the current history h is wellformed. The second branch introduces a constant u which represents the generated future as the placeholder for the method result. The left side of the implication ensures that u is fresh in h and updates the history by appending the *invocation event* for the asynchronous method call. We refer to [16] for the other ABSDL rules as well as soundness and completeness proofs of the ABSDL calculus.

```
type Pos = Pair<Int, Int>; // (x,y) coordinates
type Packet = Pair<Int, Pos>; // (id, destination)
type Buffer = Int;
data Direction = N | W | S | E | NONE;
            // north, west, south, east, the direction for not moving
data Port = P(Bool inState , Bool outState, Router rId, Buffer buff);
            // (input port state, output port state, neighbor router id, buffer size)
type Ports = Map<Direction, Port>;
```

Fig. 4. ADTs for the ASPIN model in ABS

5 Network-on-Chip Case Study

Network-on-Chip (NoC) [27] is a packet switching platform for single chip systems which scales well to an arbitrary number of resources (e.g., CPU, memory, etc.). The NoC architecture is an $m \times n$ mesh of switches and resources which are placed on the slots formed by the switches. The NoC architecture essentially is the on-chip communication infrastructure. ASPIN (Asynchronous Scalable Packet Switching Integrated Network) [32] is an example of a NoC with routers and processors. ASPIN has physically distributed routers in each core. Each router is connected to four other neighboring routers and each core is locally connected to one router. ASPIN routers are split into five separate modules (north, south, east, west, and local) with ports that have input and output channels and buffers. ASPIN uses the storage strategy of input buffering, and each input channel is provided with an independent FIFO buffer. Packets arriving from different neighboring routers (and from the local core) are stored in the respective FIFO buffer. Communication between routers is established using a four-phase handshake protocol. The protocol uses request and acknowledgment messages between neighboring routers to transfer a packet. ASPIN uses the distributed X-first algorithm to route packets from input channels to output channels. Using this algorithm, packets move along the X (horizontal) direction in the grid first, and afterwards along the Y (vertical) direction to reach their destination. The X-first algorithm is claimed to be deadlock-free [32]. In this section we model the functionality and routing algorithm of ASPIN in ABS. As a starting point we use the ASPIN model by Sharifi *et al.* [30, 31]. In Sect. 6 we will formally verify our model in ABSDL.

We model each router as an object that communicates with other routers through asynchronous method calls. The abstract data types used in our model are given in Fig. 4. We abstract away from the local communication to cores, so each router has four ports and each port has an input and output channel, the identifier rId of the neighbor router and a buffer. Packets are modeled as pairs that contain the packet identifier and the final destination coordinate.

The ABS model of a router is given in Fig. 5. The method setPorts initializes all the ports in a router and connects it with the corresponding neighbor routers. Packets are transferred using a protocol expressed in our model with two methods redirectPk and getPk. The internal method redirectPk is called when a router

```
interface Router{
    Unit setPorts(Router e, Router w, Router n, Router s);
    Unit getPk(Packet pk, Direction srcPort);}

class RouterImp(Pos address, Int buffSize) implements Router {
  Ports ports = EmptyMap;
  Set<Packet> receivedPks = EmptySet; // received packages

  Unit setPorts(Router e, Router w, Router n, Router s){
    ports = map[Pair(N, P(True, True, n, 0)), Pair(S, P(True, True, s, 0)),
                 Pair(E, P(True, True, e, 0)), Pair(W, P(True, True, w, 0))];}

  Unit getPk(Packet pk, Direction srcPort){
    if (addressPk(pk) != address) {
      await buff(lookup(ports,srcPort)) < buffSize;
      ports = put(ports,srcPort,increaseBuff(lookup(ports,srcPort)));
      this!redirectPk(pk,srcPort);}
    else { // record that packet was successfully received
      receivedPks = insertElement(receivedPks, pk); } }

  Unit redirectPk(Packet pk, Direction srcPort){
    Direction direc = xFirstRouting(addressPk(pk), address);
    await (inState(lookup(ports,srcPort)) == True)
           && (outState(lookup(ports,direc)) == True);
    ports = put(ports, srcPort, inSet(lookup(ports, srcPort), False));
    ports = put(ports, direc, outSet(lookup(ports, direc), False));
    Router r = rId(lookup(ports, direc));
    Fut<Unit> f = r!getPk(pk, opposite(direc)); await f?;
    ports = put(ports, srcPort, decreaseBuff(lookup(ports, srcPort)));
    ports = put(ports, srcPort, inSet(lookup(ports, srcPort), True));
    ports = put(ports, direc, outSet(lookup(ports, direc), True));}}
```

Fig. 5. A model of an ASPIN router using ABS

wants to redirect a packet to a neighbor router. The X-first routing algorithm in Fig. 6 decides which port direc (and as a consequence which neighbor router) to choose. The parameter srcPort determines in which input buffer the packet is temporarily and locally stored. As part of the communication protocol, the input channel of srcPort and the output channel of direc are blocked until the neighbor router confirms that it has gotten the packet, using f = r!getPk(...); **await** f? statements to simulate request and acknowledgment messages (here r is the Id of the neighbor router). The method getPk checks if the final destination of the packet is the current router, if so, it stores the packet, otherwise it temporarily stores the packet in the srcPort buffer and redirects it. The model uses standard library functions for maps and sets (e.g, put, lookup, etc.) and observers as well as other functions over the ADTs (e.g., addressPk, inState, decreaseBuff, etc.). Fig. 7 depicts a scenario with a 2×2 ASPIN chip. The sequence diagram shows how the different methods in the different routers are distributively called when a packet is sent from router R00 to router R11.

Simulation. The behavior of the ASPIN model in ABS can be analyzed using simulations. The operational semantics of ABS [23, 25] has been specified in rewriting logic which allows ABS models to be analyzed using rewriting tools.

```
def Direction xFirstRouting(Pos destination, Pos current) =
case x(current) < x(destination) {
  True => E;
  False => case x(current) > x(destination) {
              True => W;
              False => case y(current) < y(destination) {
                          True => S;
                          False => case y(current) > y(destination) {
                                      True => N;
                                      False => NONE; }; }; }; };
```

Fig. 6. X-first routing algorithm in ABS

A simulation tool for ABS based on Maude [10] is part of the ABS tool set [34]. Given an initial configuration of a 4×4 mesh, we have executed test cases where: (1) a router is the destination of its own generated packet, (2) successful arrival of packets between two neighboring routers which send packets to each other, and (3) many packets sent through the same port at the same time.

6 Formal Specification and Verification of the Case Study

In this section we formalize and verify global safety properties about our ABS NoC model in ABSDL using the KeY-ABS verification tool. This excludes any possibility of error at the level of the ABS model. Central to our verification effort are communication histories that abstractly capture the system state at any point in time [11]. Specifically, partial correctness properties are specified by finite initial segments of communication histories of the system under verification. A *history invariant* is a predicate over communication histories which holds for all finite sequences in the (prefix-closed) set of possible histories, thus expressing safety properties [3]. Our verification approach uses local reasoning about *RouterImp* objects and establishes a system invariant over the global history from invariants over the local histories of each object.

6.1 Local Reasoning

Object-oriented programming supports modular design by providing classes as the basic modular unit. Our four event semantics (described in Sect. 3) keeps the local histories of different objects disjoint, so it is possible to reason locally about each object. For ABS programs, the class invariants must hold after initialization of all class instances, must be maintained by all methods and they must hold at all process release points so that they can serve as a method contracts. We present the class invariants for RouterImp in Lemma 1 and 2 and we show the proof obligations verified by KeY-ABS that result from the reasoning of our model against the class invariants. Fig. 8 illustrates the explanations.

Lemma 1. *Whenever a router R terminates an execution of the* getPk *method, then R must either have sent an internal invocation to redirect the packet or have stored the packet in its* receivedPks *set.*

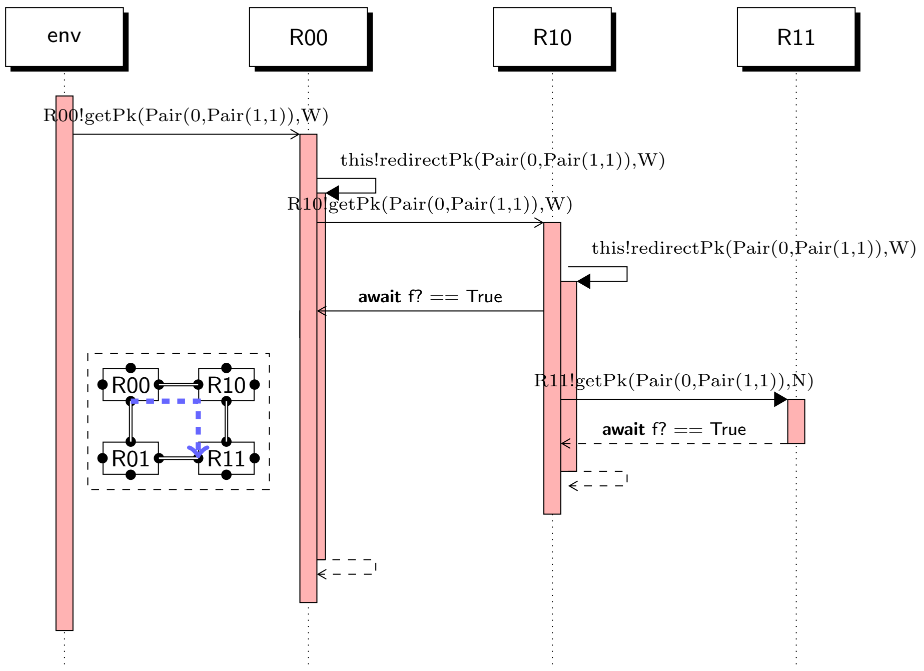

env
R00
R10
R11
R00!getPk(Pair(0,Pair(1,1)),W)
this!redirectPk(Pair(0,Pair(1,1)),W)
R10!getPk(Pair(0,Pair(1,1)),W)
this!redirectPk(Pair(0,Pair(1,1)),W)
await f? == True
R11!getPk(Pair(0,Pair(1,1)),N)
await f? == True
R00
R10
R01
R11

Fig. 7. A sequence diagram for a 2×2 ASPIN chip sending a packet to router R11

We formalize this lemma as an ABSDL formula (slightly beautified):

$$\begin{array}{l} \forall i_1, u \,.\, 0 \leq i_1 < len(h) \wedge futEv(this, u, \mathsf{getPk}, \_) = at(h, i_1) \\ \Rightarrow \\ \quad \exists i_2, pk \,.\, 0 \leq i_2 < i_1 \wedge invREv(\_, this, u, \mathsf{getPk}, (pk, \_)) = at(h, i_2) \wedge \\ \quad ((dest(pk) \neq address(this) \Rightarrow \\ \qquad \exists i_3 \,.\, i_2 < i_3 < i_1 \wedge invEv(this, this, \_, \mathsf{redirectPk}, (pk, \_)) = at(h, i_3)) \vee \\ \quad (dest(pk) = address(this) \Rightarrow pk \in \mathsf{receivedPks})) \end{array}$$

where "_" denotes a value that is of no interest. The function $len(s)$ returns the length of the sequence s, the function $at(s, i)$ returns the element located at the index i of the sequence s, the function $dest(pk)$ returns the destination address of the packet pk, and $address(r)$ returns the address of the router r.

This formula expresses that for every *future event* ev_1 of getPk with future identifier u found in history h we can find by pattern matching with u in the preceding history a corresponding *invocation reaction event* ev_2 that contains the sent packet pk. If *this* router is the destination of pk, then pk must be in its receivedPks set, otherwise an *invocation event* of redirectPk containing pk must be found in the history between events ev_1 and ev_2.

Remark 1. In the heap model of KeY-ABS, any value stored in the heap can be potentially modified while a process is released. Therefore, to prove the above

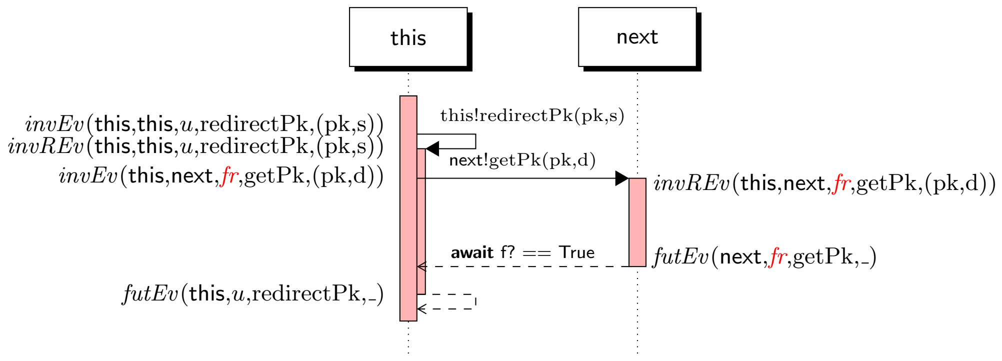

this
next
invEv(this,this,u,redirectPk,(pk,s))
invREv(this,this,u,redirectPk,(pk,s))
invEv(this,next,fr,getPk,(pk,d))
this!redirectPk(pk,s)
next!getPk(pk,d)
invREv(this,next,fr,getPk,(pk,d))
await f? == True
futEv(next,fr,getPk,_)
futEv(this,u,redirectPk,_)

Fig. 8. Communication history between a router and its neighboring router **next** where the package is sent to

property we need a somewhat stronger invariant expressing that the address of a router stored in the heap is rigid (cannot be modified by any other process). Due to a current technical limitation, we proved the invariant for a slightly simplified version of the model where the router address is passed as a parameter of getPk. This technical modification does obviously not affect the overall behavior of the model and will be lifted in future work.

Lemma 2. *Whenever a router R terminates an execution of* redirectPk*, the input channel of* srcPort *and the output channel of* direc *are released.*

Again, we formalize this lemma as an ABSDL formula:

$$\begin{gathered}\forall u \,.\, \mathit{futEv}(\mathit{this}, u, \mathsf{redirectPk}, \_) = \mathit{at}(h, \mathit{len}(h) - 1) \\ \Rightarrow \\ \exists i_1, i_2, \mathit{pk}, \mathsf{srcP}, \mathsf{dirP} \,.\, 0 < i_1 < i_2 < \mathit{len}(h) - 1 \wedge \\ (\mathit{invREv}(\mathit{this}, \mathit{this}, u, \mathsf{redirectPk}, (\mathit{pk}, \mathsf{srcP})) = \mathit{at}(h, i_1) \wedge \\ \mathit{invEv}(\mathit{this}, \_, \_, \mathsf{getPk}, (\mathit{pk}, \mathit{opposite}(\mathsf{dirP}))) = \mathit{at}(h, i_2)) \wedge \\ (\mathit{inState}(\mathit{lookup}(\mathit{ports}, \mathsf{srcP})) \wedge \mathit{outState}(\mathit{lookup}(\mathit{ports}, \mathsf{dirP})))\end{gathered}$$

This formula expresses that whenever the last event in the history h is a *future event* of redirectPk method, by pattern matching with the same future and packet in the previous history, we can find the corresponding *invocation reaction event* and the *invocation event*. In these two events we filter out the source port *srcP* and the direction port *dirP* used in the latest run of redirectPk. The input channel of srcP and the output channel of dirP must be released in the current state. This invariant captures the properties of the current state and is prefix-closed. With KeY-ABS we proved that the RouterImp class of our model satisfies this invariant.

The statistics of verifying these two invariants is given below. For each of the three methods of the RouterImp class we show it satisfies both invariants.

# nodes – # branches	**setPorts**	**getPk**	**redirectPk**
Lemma. 1	1638–12	11540–108	27077–200
Lemma. 2	214–1	1845–11	4634–34

6.2 System Specification

The global history of the system is composed of the local histories of each instance of each class. However, communication between asynchronous concurrent objects is performed by asynchronous method calls, so messages may in general be delayed in the network. The observable behavior of a system includes the possibility that the order of messages received by the callee is different from the order of messages sent by the caller. The necessary assumptions about message ordering in our setting are captured by a global notion of wellformed history.

Lemma 3. *The global history H of a system S modeled with ABS and derived from its operational semantics, is wellformed, i.e., the predicate* $\texttt{wf}(H)$ *holds.*

The formal definition of `wf` and a proof of the lemma are in [17], an informal definition is given in Sect. 3 above (see also Fig. 3).

Let $I_{this}(h)$ be the conjunction of $I_{getPk}(this, h)$ and $I_{redirectPk}(this, h)$, which are the class invariants defined in Lemma 1 and 2 where h is the local history of *this*. The local histories represent the activity of each concurrent object. We formulate a system invariant by the conjunction of the instantiated class invariants of all RouterImp objects r:

$$I(H) \triangleq \texttt{wf}(H, new_{ob}(H)) \qquad \bigwedge_{(r:\mathsf{RouterImp}) \in new_{ob}(H)} \qquad I_r(H/r)$$

where H is the global history of the system and $I_r(H/r)$ is the object invariant of r instantiated from the class invariant $I_{this}(h)$. The local history of r is obtained by the projection H/r from the global history. The function $new_{ob}(H)$ returns the set of RouterImp objects generated within the system execution captured by H. Each wellformed interleaving of the local histories represents a possible global history. History wellformedness $\texttt{wf}(H, new_{ob}(H))$ ensures proper ordering of those events that belong to the same method invocation. The composition rule was proved sound in [16]. As a consequence, we obtain:

Theorem 1. *Whenever a router R releases a pair of input and output channels used for redirecting a receiving packet, the next router of R must either have sent an internal invocation to redirect the packet or have stored the packet in its receivedPks set. Hence, the network does not drop any packets.*

Effort. The modelling of the NoC case study in ABS took ca. two person weeks. Formal specification and verification was mainly done by the first author of this paper who at the time was not experienced with the verification tool KeY-ABS. The effort for formal specification was ca. two person weeks and for formal verification ca. one person month, but this included training to use the tool effectively. For an experienced user of KeY-ABS, we estimate that these figures would be three person days and one person week, respectively.

7 Future Work

Deadlock Analysis. In addition to history-based invariants, it is conceivable to prove other properties, such as deadlock-freedom. Deadlocks may occur in a system, for example, when a shared buffer between processes is full and one process can decrease the buffer size only if the other process increases the buffer size. This situation is prevented in the ABS model by disallowing self-calls before decreasing the size of the buffer (the method invocation of *getPk* within *redirectPk* in our model is an external call). It is possible to argue informally that our ABS model of NoC is indeed deadlock-free, but a formal proof with KeY-ABS is future work. The main obstacle is that deadlocks are a global property and one would need to find a way to encode sufficient conditions for deadlock-freedom into the local histories. There are deadlock analyzers for ABS [20], but these, like other approaches to deadlock analysis of concurrent systems, work only for a fixed number of objects.

Extensions of the Model. The ASPIN chip model presented in this paper can easily be extended with time (e.g, delays and deadline annotations) and scheduling (e.g., FIFO, EDF, user-defined, etc.) using Real-Time ABS [6]. The extension with time would allow us to run simulations and obtain results about the performance of the model. Adding scheduling to the model would allow us to, for example, guarantee the ordering of the sent packets (using FIFO scheduling) or to express priority of packets. We can also easily change the routing algorithm in Fig. 6 without any need to alter the RouterImp class in Fig. 5. It is possible to compare the performance of different routing algorithms by means of simulations.

Runtime Assertion Checking. Another extension to the model could be runtime assertion checking (RAC) [18], for example, to ensure that packets make progress towards their final destination. For this one would use the distance function in Fig. 9 and simply include the assertion into the model, where addr is the address of the current router and prevAddr is the address of the previous neighbor router from where the packet was redirected. RAC is already supported by the ABS tool set and can be used for this case study, but to keep the paper focussed we decided not to report the results here.

8 Related Work and Conclusion

Previous work on formal modeling of NoC includes [8, 12, 30, 31]. The papers [30, 31], which were a starting point for our work, present a formal model of NoC in the actor-based modeling language Rebeca [26, 33]. In [30], the authors

```
def Int distance(Pos destination, Pos current) =
    abs(x(destination) − x(current)) + abs(y(destination) − y(current));

assert (distance(addressPk(pk),addr)==distance(addressPk(pk),prevAddr)−1);
```

Fig. 9. A function to calculate the distance between the current position and the final destination of a packet for the X-first routing algorithm

model the functional and timed behavior of ASPIN (with the X-first routing algorithm). To analyze their model, they used the model checker of Rebeca, Afra [26], to guarantee deadlock-freedom and successful packet sending for *specific* chip configurations. They also measure the maximum end-to-end latency of packets. In [31] the authors compare the performance of different routing algorithms. The ASPIN model presented in this paper does not capture timing behavior and uses the X-first routing algorithm, but timing behavior can easily be added and other routing algorithms can be plugged into the model as explained in Sect. 7. Compared to the Rebeca model, our ABS model of the ASPIN chip is deadlock-free and more compact. It is decoupled from the routing algorithm and easier to understand than the Rebeca model, because ABS permits intuitive, object-oriented modeling of the involved concepts, as well as high-level concepts for modeling concurrency. Our verification approach deals with an *unbounded* number of objects and is valid for *generic* NoC models for any $m \times n$ mesh in the ASPIN chip as well as any number of sent packets. This is possible, because we use *deductive verification* in the expressive program logic ABSDL with the verification tool KeY-ABS [7, 16] and formal specification of observable behavior [14, 15]. This allowed us to prove *global safety properties* of the system using *local* rules and symbolic execution. In contrast to model checking this allows us to deal effectively with unbounded target systems without encountering state explosion.

Acknowledgements. The authors gratefully acknowledge valuable discussions with Richard Bubel.

References

1. G. A. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems.* The MIT Press, Cambridge, Mass., 1986.
2. E. Albert, F. S. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte, S. L. Tapia Tarifa, and P. Y. H. Wong. Formal modeling of resource management for cloud architectures: An industrial case study using Real-Time ABS. *Journal of Service-Oriented Computing and Applications*, 8(4):323–339, Dec. 2014.
3. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, Oct. 1985.
4. J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.

5. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.
6. J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 9(1):29–43, 2013.
7. R. Bubel, A. Flores Montoya, and R. Hähnle. Analysis of executable software models. In M. Bernardo, F. Damiani, R. Hähnle, E. B. Johnsen, and I. Schaefer, editors, *Executable Software Models: 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM'14)*, volume 8483 of *LNCS*, pages 1–27. Springer, June 2014.
8. Y.-R. Chen, W.-T. Su, P.-A. Hsiung, Y.-C. Lan, Y.-H. Hu, and S.-J. Chen. Formal modeling and verification for network-on-chip. In *International Conference on Green Circuits and Systems (ICGCS 2010)*, pages 299–304. IEEE Computer Society Press, 2010.
9. D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, and P. Y. H. Wong. Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In M. Bernardo and V. Issarny, editors, *Proc. 11th Intl. School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM 2011)*, volume 6659 of *LNCS*, pages 417–457. Springer, 2011.
10. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, Aug. 2002.
11. O.-J. Dahl. *Verifiable Programming.* International Series in Computer Science. Prentice Hall, New York, N.Y., 1992.
12. S. Dasgupta. *Formal design and synthesis of GALS architectures.* PhD thesis, University of Newcastle, January 2008.
13. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. de Nicola, editor, *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 316–330. Springer, Mar. 2007.
14. C. C. Din, J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. *Journal of Logic and Algebraic Programming*, 81(3):227–256, 2012.
15. C. C. Din, J. Dovland, and O. Owe. Compositional reasoning about shared futures. In *10th International Conference on Software Engineering and Formal Methods (SEFM 2012)*, volume 7504 of *LNCS*, pages 94–108. Springer, 2012.
16. C. C. Din and O. Owe. Compositional reasoning about active objects with shared futures. *Formal Aspects of Computing*, Nov. 2014. Available online.
17. C. C. Din and O. Owe. A sound and complete reasoning system for asynchronous communication with shared futures. *Journal of Logical and Algebraic Methods in Programming*, 83(5–6):360–383, 2014.
18. C. C. Din, O. Owe, and R. Bubel. Runtime assertion checking and theorem proving for concurrent and distributed systems. In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2014)*, pages 480–487. SciTePress, 2014.
19. J. Dovland, E. B. Johnsen, and O. Owe. Verification of concurrent objects with asynchronous method calls. In *Proc. IEEE Intl. Conference on Software Science, Technology & Engineering(SwSTE'05)*, pages 141–150. IEEE Computer Society Press, Feb. 2005.
20. E. Giachino, C. Laneve, and M. Lienhardt. A framework for deadlock detection in ABS. *Software and Systems Modeling*, 2015. To appear.

21. R. Hähnle, M. Helvensteijn, E. B. Johnsen, M. Lienhardt, D. Sangiorgi, I. Schaefer, and P. Y. H. Wong. HATS abstract behavioral specification: The architectural view. In B. Beckert, F. Damiani, F. S. de Boer, and M. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 7542 of *LNCS*, pages 109–132. Springer, 2013.
22. C. A. R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice Hall, 1985.
23. 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, 2011.
24. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
25. E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Integrating deployment architectures and resource consumption in timed object-oriented models. *Journal of Logical and Algebraic Methods in Programming*, 84(1):67–91, 2015.
26. E. Khamespanah, Z. Sabahi-Kaviani, R. Khosravi, M. Sirjani, and M. Izadi. Timed-rebeca schedulability and deadlock-freedom analysis using floating-time transition system. In *Proceedings of the 2nd edition of Programming systems, languages and applications based on actors, agents, and decentralized control abstractions, (AGERE! 2012)*, pages 23–34. ACM, 2012.
27. S. Kumar, A. Jantsch, M. Millberg, J. Öberg, J. Soininen, M. Forsell, K. Tiensyrjä, and A. Hemani. A network on chip architecture and design methodology. In *2002 IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2002), 25-26 April 2002, Pittsburgh, PA, USA*, pages 117–124, 2002.
28. B. C. Pierce. *Types and Programming Languages.* The MIT Press, 2002.
29. J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *European Conference on Object-Oriented Programming (ECOOP 2010)*, volume 6183 of *LNCS*, pages 275–299. Springer, June 2010.
30. Z. Sharifi, S. Mohammadi, and M. Sirjani. Comparison of NoC routing algorithm using formal methods. In H. R. Arabnia, H. Ishii, M. Ito, K. Joe, H. Nishikawa, F. G. Tinetti, G. A. Gravvanis, G. Jandieri, and A. M. G. Solo, editors, *Proc. of Parallel and Distributed Processing Techniques and Applications (PDPTA 2013)*, volume 2, pages 474–482. CSREA Press, 2013.
31. Z. Sharifi, M. Mosaffa, S. Mohammadi, and M. Sirjani. Functional and performance analysis of Network-on-Chips using Actor-based modeling and formal verification. *ECEASST*, 66, 2013.
32. A. Sheibanyrad, A. Greiner, and I. M. Panades. Multisynchronous and fully asynchronous NoCs for GALS architectures. *IEEE Design & Test of Computers*, 25(6):572–580, 2008.
33. M. Sirjani and M. M. Jaghoori. Ten years of analyzing actors: Rebeca experience. In *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*, volume 7000 of *LNCS*, pages 20–56. Springer, 2011.
34. 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. *STTT*, 14(5):567–588, 2012.

Appendix D

Deadlock Analysis

Softw Syst Model
DOI 10.1007/s10270-014-0444-y

THEME SECTION PAPER

A framework for deadlock detection in `core ABS`

Elena Giachino · Cosimo Laneve · Michael Lienhardt

Received: 15 November 2013 / Revised: 16 July 2014 / Accepted: 3 December 2014

Abstract We present a framework for statically detecting deadlocks in a concurrent object-oriented language with asynchronous method calls and cooperative scheduling of method activations. Since this language features recursion and dynamic resource creation, deadlock detection is extremely complex and state-of-the-art solutions either give imprecise answers or do not scale. In order to augment precision and scalability, we propose a modular framework that allows several techniques to be combined. The basic component of the framework is a front-end inference algorithm that extracts abstract behavioral descriptions of methods, called contracts, which retain resource dependency information. This component is integrated with a number of possible different back-ends that analyze contracts and derive deadlock information. As a proof-of-concept, we discuss two such back-ends: (1) an evaluator that computes a fixpoint semantics and (2) an evaluator using abstract model checking.

Keywords Type inference · Deadlock analysis · Asynchronous method invocation · Concurrent object groups

1 Introduction

Modern systems are designed to support a high degree of parallelism by letting as many system components as possible operate concurrently. When such systems also exhibit a high degree of resource and data sharing, then deadlocks represent an insidious and recurring threat. In particular, 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 with unbounded (mutual) recursion and dynamic resource creation. A paradigm case is an adaptive system that creates an unbounded number of processes such as server applications. In these systems, the interaction protocols are extremely complex and state-of-the-art solutions either give imprecise answers or do not scale—see Sect. 8, and for instance, Naik et al. [31] and the references therein.

In order to augment precision and scalability, we propose a modular framework that allows several techniques to be combined. We meet scalability requirement by designing a front-end inference system that automatically extracts abstract behavioral descriptions pertinent to deadlock analysis, called *contracts*, from code. The inference system is *modular* because it (partially) supports separate inference of modules. To meet precision of contracts' analysis, as a proof-of-concept, we define and implement two different techniques: (1) an evaluator that computes a fixpoint semantics and (2) an evaluator using abstract model checking.

Our framework targets `core ABS` [22], which is an abstract, executable, object-oriented modeling language with a formal semantics, targeting distributed systems. In `core ABS`, method invocations are asynchronous: the caller continues after the invocation and the called code runs on

Communicated by Prof. Einar Broch Johnsen and Luigia Petre.

Partly funded by the EU Project FP7-610582 ENVISAGE: Engineering Virtualized Services.

E. Giachino (✉) · C. Laneve · M. Lienhardt
Department of Computer Science and Engineering,
University of Bologna – INRIA Focus Team, Bologna, Italy
e-mail: elena.giachino@unibo.it

Published online: 29 January 2015

a different task. Tasks are cooperatively scheduled, that is, there is a notion of group of objects, called *cog*, and there is at most one active task at each time per cog. The active task explicitly returns the control in order to let other tasks progress. The synchronization between the caller and the called methods is performed when the result is strictly necessary [4,23,40]. Technically, the decoupling of method invocation and the returned value is realized using *future variables* (see [9] and the references in there), which are pointers to values that may be not available yet. Clearly, the access to values of future variables may require waiting for the value to be returned. We discuss the syntax and the semantics of `core ABS`, in Sect. 2.

Because of the presence of explicit synchronization operations, the analysis of deadlocks in `core ABS` is more fine-grained than in thread-based languages (such as `Java`). However, as usual with (concurrent) programming languages, analyses are hard and time-consuming because most part of the code is irrelevant for the properties one intends to derive. For this reason, in Sect. 4, we design an inference system that *automatically extracts contracts* from `core ABS` code. These contracts are similar to those ranging from languages for session types [13] to process contracts [28] and to calculi of processes as Milner's CCS or pi-calculus [29,30]. The inference system mostly collects method behaviors and uses constraints to enforce consistencies among behaviors. Then, a standard semiunification technique is used for solving the set of generated constraints.

Since our inference system addresses a language with asynchronous method invocations, it is possible that a method triggers behaviors that will last *after* its lifetime (and therefore will contribute to *future* deadlocks). In order to support a more precise analysis, we split contracts of methods in *synchronized* and *unsynchronized contracts*, with the intended meaning that the formers collect the invocations that are explicitly synchronized in the method body and the latter ones collect the other invocations.

The current release of the inference system does not cover the full range of features of `core ABS`. In Sect. 3, we discuss the restrictions of `core ABS` and the techniques that may be used to remove these restrictions.

Our contracts feature recursion and resource creation; therefore, their underlying models contain infinite states and their analysis cannot be exhaustive. We propose two techniques for analyzing contracts (and to show the modularity of our framework). The first one, which is discussed in Sect. 5, is 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, this technique may signal false positives). The second technique, which is detailed in Sect. 6, is an abstract model checking that evaluates the contract program upto some point, which is possible to determine by analyzing the recursive patterns of the program. This technique is precise when the recursions are linear, while it is over-approximating in general.

We have prototyped an implementation of our framework, called the `DF4ABS` tool, and in Sect. 7, we assess the precision and performance of the prototype. In particular, we have applied it to an industrial case study that is based on the Fredhopper Access Server (FAS) developed by SDL Fredhopper.[1] It is worth to recall that, because of the modularity of `DF4ABS`, the current analyses techniques may be integrated and/or replaced by other ones. We discuss this idea in Sect. 9.

1.1 Origin of the material

The basic ideas of this article have appeared in conference proceedings. In particular, the contract language and (a simplified form of) the inference system have been introduced in [14,16], while the fixpoint analysis technique has been explored in [14], and an introduction to the abstract model-checking technique can be found in [17], while the details are in [18]. This article is a thoroughly revised and enhanced version of [14] that presents the whole framework in a uniform setting and includes the full proofs of all the results. A more detailed comparison with other related work is postponed to Sect. 8.

2 The language `core ABS`

The syntax and the semantics (of the concurrent object level) of `core ABS` are defined in the following two subsections; the third subsection is devoted to the discussion of examples, and the last one to the definition of deadlock. In this contribution, we overlook the functional level of `core ABS` that defines data types and functions because their analysis can be performed with techniques that may (easily) complement those discussed in this paper (such as data-flow analysis). Details of `core ABS`, its semantics and its standard type system can be also found in [22].

2.1 Syntax

Figure 1 displays `core ABS` syntax, where an overlined element corresponds to any finite sequence of such element. The elements of the sequence are separated by commas, except for $\overline{C}$, which has no separator. For example, $\overline{T}$ means a (possibly empty) sequence $T_1, \ldots, T_n$. When we write $\overline{T\ x\ ;}$ we mean a sequence $T_1\ x_1;\ \cdots;\ T_n\ x_n;$ when the sequence is not empty; we mean the empty sequence otherwise.

A program P is a list of interface and class declarations (resp. I and C) followed by a *main function* $\{\ \overline{T\ x\ ;}\ s\ \}$. A

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

$$\begin{array}{rcll}
P & ::= & \overline{I}\ \overline{C}\ \{\,\overline{T\ x\ ;}\ s\,\} & \text{program}\\
T & ::= & \mathtt{D}\ \mid\ \mathtt{Fut<}T\mathtt{>}\ \mid\ \mathtt{I} & \text{type}\\
I & ::= & \mathtt{interface\ I}\ \{\,\overline{S\ ;}\,\} & \text{interface}\\
S & ::= & T\ \mathtt{m}(\overline{T\ x}) & \text{method signature}\\
C & ::= & \mathtt{class\ C}(\overline{T\ x})\ [\mathtt{implements}\ \overline{\mathtt{I}}]\ \{\,\overline{T'\ x';}\ \overline{M}\,\} & \text{class}\\
M & ::= & S\{\,\overline{T\ x\ ;}\ s\,\} & \text{method definition}\\
s & ::= & \mathtt{skip}\ \mid\ x = z\ \mid\ \mathtt{if}\ e\ \{\,s\,\}\ \mathtt{else}\ \{\,s\,\}\ \mid\ \mathtt{return}\ e\ \mid\ s\ ;\ s\ \mid\ \mathtt{await}\ e? & \text{statement}\\
z & ::= & e\ \mid\ e\mathtt{.m}(\overline{e})\ \mid\ e\mathtt{!m}(\overline{e})\ \mid\ \mathtt{new}\ \ \mathtt{C}\ (\overline{e})\ \mid\ \mathtt{new\ cog\ C}\ (\overline{e})\ \mid\ e\mathtt{.get} & \text{expression with side effects}\\
e & ::= & v\ \mid\ x\ \mid\ \mathtt{this}\ \mid\ \textit{arithmetic-and-bool-exp} & \text{expression}\\
v & ::= & \mathtt{null}\ \mid\ \textit{primitive values} & \text{value}
\end{array}$$

Fig. 1 Language core ABS

type T is the name of either a primitive type D such as Int, Bool, String, or a *future type* Fut<T>, or an interface name I.

A class declaration class C$(\overline{T\ x})$ $\{\overline{T'\ x'\ ;}\ \overline{M}\}$ has a name C and declares its fields $\overline{T\ x}$, $\overline{T'\ x'}$ and its methods $\overline{M}$. The fields $\overline{T\ x}$ will be initialized when the object is created; the fields $\overline{T'\ x'}$ will be initialized by the main function of the class (or by the other methods).

A statement s may be either one of the standard operations of an imperative language or one of the operations for scheduling. This operation is await x? (the other one is get, see below), which suspends method's execution until the argument x, is resolved. This means that await requires the value of x to be resolved before resuming method's execution.

An expression z may have side effects (may change the state of the system) and is either an object creation new C$(\bar{e})$ in the same group of the creator or an object creation new cog C$(\bar{e})$ in a new group. In core ABS, (runtime) objects are partitioned in groups, called *cogs*, which own a lock for regulating the executions of threads. Every thread acquires its own cog lock in order to be evaluated and releases it upon termination or suspension. Clearly, threads running on different cogs may be evaluated in parallel, while threads running on the same cog do compete for the lock and interleave their evaluation. The two operations new C$(\bar{e})$ and new cog C$(\bar{e})$ allow one to add an object to a previously created cog or to create new singleton cogs, respectively.

An expression z may also be either a (synchronous) method call e.m$(\bar{e})$ or an *asynchronous* method call e!m$(\bar{e})$. Synchronous method invocations suspend the execution of the caller, without releasing the lock of the corresponding cog; asynchronous method invocations do not suspend caller's execution. Expressions z also include the operation e.get that suspends method's execution until the value of e is computed. The type of e is a future type that is associated with a method invocation. The difference between await x? and e.get is that the former releases cog's lock when the value of x is still unavailable; the latter does not release cog's lock (thus being the potential cause of a deadlock).

A *pure* expression e is either a value, or a variable x, or the reserved identifier this. Values include the null object and primitive type values, such as true and 1.

In the whole paper, we assume that sequences of field declarations $\overline{T'\ x'}$, method declarations $\overline{M}$ and parameter declarations $\overline{T\ x}$ do not contain duplicate names. It is also assumed that every class and interface name in a program has a unique definition.

2.2 Semantics

core ABS semantics is defined as a transition relation between *configurations*, noted *cn* and defined in Fig. 2. Configurations are sets of elements—therefore, we identify configurations that are equal upto associativity and commutativity—and are denoted by the juxtaposition of the elements *cn cn*; the empty configuration is denoted by ε. The transition relation uses three infinite sets of names: *object names*, ranged over by o, o', ..., *cog names*, ranged over by c, c', ..., and *future names*, ranged over by f, f', ... Object names are partitioned according to the class and the cog they belong. We assume there are infinitely many object names per class, and the function fresh(C) returns a new object name of class C. Given an object name o, the function class(o) returns its class. The function fresh( ) returns either a fresh cog name or a fresh future name; the context will disambiguate between the twos.

Runtime values are either values v in Fig. 1 or object and future names or an undefined value, which is denoted by $\perp$.

Runtime statements extend normal statements with cont(f) that is used to model explicit continuations in synchronous invocations. With an abuse of notation, we range over runtime values with v, v', ... and over runtime statements with s, s', ... We finally use a and l, possibly indexed, to range over maps from fields to runtime values and local variables to runtime values, respectively. The map l also binds the special name destiny to a future value.

The elements of configurations are

– *objects* $ob(o, a, p, q)$ where o is an object name; a returns the values of object's fields, p is either idle, rep-

$$\begin{array}{rl} cn ::= & \epsilon \mid fut(f, val) \mid ob(o, a, p, q) \mid invoc(o, f, \mathtt{m}, \overline{v}) \mid cog(c, act) \mid cn\ cn \\ p ::= & \{l \mid s\} \mid \mathtt{idle} \\ q ::= & \epsilon \mid \{l \mid s\} \mid q\ q \\ s ::= & \mathtt{cont}(f) \mid \ldots \end{array} \qquad \begin{array}{rl} act ::= & o \mid \varepsilon \\ val ::= & v \mid \bot \\ a ::= & [\cdots, x \mapsto v, \cdots] \\ v ::= & o \mid f \mid \ldots \end{array}$$

Fig. 2 Runtime syntax of `core ABS`

resenting inactivity, or is the *active process* $\{l \mid s\}$, where l returns the values of local identifiers and s is the statement to evaluate; q is a set of processes to evaluate.

– *future binders* $fut(f, v)$ where v, called *the reply value* may be also $\bot$ meaning that the value has still not computed.
– *cog binders* $cog(c, o)$ where o is the active object; it may be ε meaning that the cog c has no active object.
– *method invocations* $invoc(o, f, \mathtt{m}, \overline{v})$.

The following auxiliary functions are used in the semantic rules (we assume a fixed `core ABS` program):

– dom(l) and dom(a) return the domain of l and a, respectively.
– $l[x \mapsto v]$ is the function such that $(l[x \mapsto v])(x) = v$ and $(l[x \mapsto v])(y) = l(y)$, when $y \neq x$. Similarly for $a[x \mapsto v]$.
– $[\![e]\!]_{(a+l)}$ returns the value of e by computing the arithmetic and Boolean expressions and retrieving the value of the identifiers that is stored either in a or in l. Since a and l are assumed to have disjoint domains, we denote the union map with $a+l$. $[\![\overline{e}]\!]_{(a+l)}$ returns the tuple of values of $\overline{e}$. When e is a future name, the function $[\![\cdot]\!]_{(a+l)}$ is the identity. Namely $[\![f]\!]_{(a+l)} = f$.
– $\mathtt{C}.\mathtt{m}$ returns the term $(\overline{T\ x})\{\overline{T'\ z}; s\}$ that contains the arguments and the body of the method $\mathtt{m}$ in the class $\mathtt{C}$.
– bind$(o, f, \mathtt{m}, \overline{v}, \mathtt{C}) = \{[\text{destiny} \mapsto f, \overline{x} \mapsto \overline{v}, \overline{z} \mapsto \bot] \mid s[^{o}/_{\mathtt{this}}]\}$, where $\mathtt{C}.\mathtt{m} = (\overline{T\ x})\{\overline{T'\ z}; s\}$.
– init$(\mathtt{C}, o)$ returns the process

  $\{\varnothing[\text{destiny} \mapsto f_\bot] \mid s[^{o}/_{\mathtt{this}}]\}$

  where $\{\overline{T\ x}; s\}$ is the main function of the class $\mathtt{C}$. The special name destiny is initialized to a fresh (future) name $f_\bot$.
– atts$(\mathtt{C}, \overline{v}, c)$ returns the map $[cog \mapsto c, \overline{x} \mapsto \overline{v}, \overline{x'} \mapsto \bot]$, where the class $\mathtt{C}$ is defined as

  $\mathtt{class\ C}(\overline{T\ x})\{\ \overline{T'\ x'\ ;}\ \ \overline{M}\ \}$

  and where cog is a special field storing the cog name of the object.

The transition relation rules are collected in Figs. 3 and 4. They define transitions of objects $ob(o, a, p, q)$ according to the shape of the statement in p. We focus on rules concerning the concurrent part of `core ABS`, since the other ones are standard. Rules (Await- True) and (Await- False) model the `await` e`?` operation: If the (future) value of e has been computed then `await` terminates, otherwise the active process becomes idle. In this case, if the object owns the control of the cog, then it may release such control—rule (Release- Cog). Otherwise, when the cog has no active process, the object gets the control of the cog and activates one of its processes—rule (Activate). Rule (Read- Fut) permits the retrieval of the value returned by a method; the object does not release the control of the cog until this value has been computed.

The two types of object creation are modeled by (New-Object) and (New- Cog- Object). The first one creates the new object in the same cog. The new object is idle because the cog has already an active object. The second one creates the object in a new cog and makes it active by scheduling the process corresponding to the main function of the class. The special field cog is initialized accordingly; the other object's fields are initialized by evaluating the arguments of the operation—see definition of atts.

Rule (Async- Call) defines asynchronous method invocation $x = e\,!\,\mathtt{m}(\overline{e})$. This rule creates a fresh future name that is assigned to the identifier x. The evaluation of the called method is transferred to a different process—see rule (Bind- Mtd). Therefore, the caller can progress without waiting for callee's termination. Rule (Cog- Sync- Call) defines synchronous method invocation on an object in the *same* cog (because of the premise $a'(\text{cog}) = c$ and the element $cog(c, o)$ in the configuration). The control is passed to the called object that executes the body of the called method followed by a special statement $\mathtt{cont}(f')$, where f' is a fresh future name. When the evaluation of the body terminates, the caller process is scheduled again using the name f'—see rule (Cog- Sync- Return- Sched). Rules (Self- Sync- Call) and (Rem- Sync- Call) deal with synchronous method invocations of the same object and of objects in different cogs, respectively. The former is similar to (Cog- Sync- Call) except that there is no control on cogs. The latter one implements the synchronous invocation through an asynchronous one followed by an explicit synchronization operation.

It is worth to observe that the rules (Activate), (Cog- Sync-Call) and (Self- Sync- Call) are different from the corresponding ones in [22]. In fact, in [22], rule (Activate) uses an unspecified *select* predicate that activates one task from the queue of processes to evaluate. According to the rules (Cog-

$$\text{(SKIP)}\quad \begin{array}{c} ob(o,a,\{l \mid \texttt{skip}; s\},q) \\ \to ob(o,a,\{l \mid s\},q) \end{array}$$

$$\text{(ASSIGN-LOCAL)}\quad \frac{x \in \mathrm{dom}(l) \qquad v = [\![e]\!]_{(a+l)}}{\begin{array}{c} ob(o,a,\{l \mid x = e; s\},q) \\ \to ob(o,a,\{l[x \mapsto v] \mid s\},q) \end{array}}$$

$$\text{(ASSIGN-FIELD)}\quad \frac{x \in \mathrm{dom}(a) \setminus \mathrm{dom}(l) \qquad v = [\![e]\!]_{(a+l)}}{\begin{array}{c} ob(o,a,\{l \mid x = e; s\},q) \\ \to ob(o,a[x \mapsto v],\{l \mid s\},q) \end{array}}$$

$$\text{(COND-TRUE)}\quad \frac{\texttt{true} = [\![e]\!]_{(a+l)}}{\begin{array}{c} ob(o,a,\{l \mid \texttt{if}\ e\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\}; s\},q) \\ \to ob(o,a,\{l \mid s_1; s\},q) \end{array}}$$

$$\text{(COND-FALSE)}\quad \frac{\texttt{false} = [\![e]\!]_{(a+l)}}{\begin{array}{c} ob(o,a,\{l \mid \texttt{if}\ e\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\}; s\},q) \\ \to ob(o,a,\{l \mid s_2; s\},q) \end{array}}$$

$$\text{(AWAIT-TRUE)}\quad \frac{f = [\![e]\!]_{(a+l)} \qquad v \neq \bot}{\begin{array}{c} ob(o,a,\{l \mid \texttt{await}\ e\,?; s\},q)\ \ fut(f,v) \\ \to ob(o,a,\{l \mid s\},q)\ \ fut(f,v) \end{array}}$$

$$\text{(AWAIT-FALSE)}\quad \frac{f = [\![e]\!]_{(a+l)}}{\begin{array}{c} ob(o,a,\{l \mid \texttt{await}\ e\,?; s\},q)\ \ fut(f,\bot) \\ \to ob(o,a,\mathrm{idle},q \cup \{l \mid \texttt{await}\ e\,?; s\})\ \ fut(f,\bot) \end{array}}$$

$$\text{(RELEASE-COG)}\quad \begin{array}{c} ob(o,a,\mathrm{idle},q)\ \ cog(c,o) \\ \to ob(o,a,\mathrm{idle},q)\ \ cog(c,\epsilon) \end{array}$$

$$\text{(ACTIVATE)}\quad \frac{c = a(\mathrm{cog})}{\begin{array}{c} ob(o,a,\mathrm{idle},q \cup \{l \mid s\})\ \ cog(c,\epsilon) \\ \to ob(o,a,\{l \mid s\},q)\ \ cog(c,o) \end{array}}$$

$$\text{(READ-FUT)}\quad \frac{f = [\![e]\!]_{(a+l)} \qquad v \neq \bot}{\begin{array}{c} ob(o,a,\{l \mid x = e\texttt{.get}; s\},q)\ \ fut(f,v) \\ \to ob(o,a,\{l \mid x = v; s\},q)\ \ fut(f,v) \end{array}}$$

$$\text{(NEW-OBJECT)}\quad \frac{\begin{array}{c} o' = \mathrm{fresh}(\texttt{C}) \qquad p = \mathrm{init}(\texttt{C},o') \\ a' = \mathrm{atts}(\texttt{C},[\![\overline{e}]\!]_{(a+l)},c) \end{array}}{\begin{array}{c} ob(o,a,\{l \mid x = \texttt{new C}(\overline{e}); s\},q)\ \ cog(c,o) \\ \to ob(o,a,\{l \mid x = o'; s\},q)\ \ cog(c,o) \\ ob(o',a',\mathrm{idle},\{p\}) \end{array}}$$

$$\text{(NEW-COG-OBJECT)}\quad \frac{\begin{array}{c} c' = \mathrm{fresh}(\,) \qquad o' = \mathrm{fresh}(\texttt{C}) \qquad p = \mathrm{init}(\texttt{C},o') \\ a' = \mathrm{atts}(\texttt{C},[\![\overline{e}]\!]_{(a+l)},c') \end{array}}{\begin{array}{c} ob(o,a,\{l \mid x = \texttt{new cog C}(\overline{e}); s\},q) \\ \to ob(o,a,\{l \mid x = o'; s\},q) \\ ob(o',a',p,\varnothing) \quad cog(c',o') \end{array}}$$

Fig. 3 Semantics of core ABS(1)

$$\text{(ASYNC-CALL)}\quad \frac{o' = [\![e]\!]_{(a+l)} \qquad \overline{v} = [\![\overline{e}]\!]_{(a+l)} \qquad f = \mathrm{fresh}(\,)}{\begin{array}{c} ob(o,a,\{l \mid x = e!\texttt{m}(\overline{e}); s\},q) \\ \to ob(o,a,\{l \mid x = f; s\},q)\ \ invoc(o',f,\texttt{m},\overline{v})\ \ fut(f,\bot) \end{array}}$$

$$\text{(BIND-MTD)}\quad \frac{\{l \mid s\} = \mathrm{bind}(o,f,\texttt{m},\overline{v},\mathrm{class}(o))}{\begin{array}{c} ob(o,a,p,q)\ \ invoc(o,f,\texttt{m},\overline{v}) \\ \to ob(o,a,p,q \cup \{l \mid s\}) \end{array}}$$

$$\text{(COG-SYNC-CALL)}\quad \frac{\begin{array}{c} o' = [\![e]\!]_{(a+l)} \qquad \overline{v} = [\![\overline{e}]\!]_{(a+l)} \qquad f = \mathrm{fresh}(\,) \\ c = a'(\mathrm{cog}) \qquad f' = l(\mathrm{destiny}) \\ \{l' \mid s'\} = \mathrm{bind}(o',f,\texttt{m},\overline{v},class(o')) \end{array}}{\begin{array}{c} ob(o,a,\{l \mid x = e.\texttt{m}(\overline{e}); s\},q) \\ ob(o',a',\mathrm{idle},q')\ \ cog(c,o) \\ \to ob(o,a,\mathrm{idle},q \cup \{l \mid \texttt{await}\ f?; x = f.\texttt{get}; s\})\ \ fut(f,\bot) \\ ob(o',a',\{l' \mid s'; \texttt{cont}\ f'\},q')\ \ cog(c,o') \end{array}}$$

$$\text{(COG-SYNC-RETURN-SCHED)}\quad \frac{c = a'(\mathrm{cog}) \qquad f = l'(\mathrm{destiny})}{\begin{array}{c} ob(o,a,\{l \mid \texttt{cont}\ f\},q)\ \ cog(c,o) \\ ob(o',a',\mathrm{idle},q' \cup \{l' \mid s\}) \\ \to ob(o,a,\mathrm{idle},q)\ \ cog(c,o') \\ ob(o',a',\{l' \mid s\},q') \end{array}}$$

$$\text{(SELF-SYNC-CALL)}\quad \frac{\begin{array}{c} f' = l(\mathrm{destiny}) \qquad o = [\![e]\!]_{(a+l)} \qquad \overline{v} = [\![\overline{e}]\!]_{(a+l)} \\ f = \mathrm{fresh}(\,) \qquad \{l' \mid s'\} = \mathrm{bind}(o,f,\texttt{m},\overline{v},class(o)) \end{array}}{\begin{array}{c} ob(o,a,\{l \mid x = e.\texttt{m}(\overline{e}); s\},q) \\ \to ob(o,a,\{l' \mid s'; \texttt{cont}(f')\},q \cup \{l \mid \texttt{await}\ f?; x = f.\texttt{get}; s\})\ \ fut(f,\bot) \end{array}}$$

$$\text{(RETURN)}\quad \frac{v = [\![e]\!]_{(a+l)} \qquad f = l(\mathrm{destiny})}{\begin{array}{c} ob(o,a,\{l \mid \texttt{return}\ e; s\},q)\ \ fut(f,\bot) \\ \to ob(o,a,\{l \mid s\},q)\ \ fut(f,v) \end{array}}$$

$$\text{(REM-SYNC-CALL)}\quad \frac{o' = [\![e]\!]_{(a+l)} \qquad f = \mathrm{fresh}(\,) \qquad a(\mathrm{cog}) \neq a'(\mathrm{cog})}{\begin{array}{c} ob(o,a,\{l \mid x = e.\texttt{m}(\overline{e}); s\},q)\ \ ob(o',a',p,q') \\ \to ob(o,a,\{l \mid f = e!\texttt{m}(\overline{e}); x = f.\texttt{get}; s\},q) \\ ob(o',a',p,q') \end{array}}$$

$$\text{(SELF-SYNC-RETURN-SCHED)}\quad \frac{f = l'(\mathrm{destiny})}{\begin{array}{c} ob(o,a,\{l \mid \texttt{cont}(f)\},q \cup \{l' \mid s\}) \\ \to ob(o,a,\{l' \mid s\},q) \end{array}}$$

$$\text{(CONTEXT)}\quad \frac{cn \to cn'}{cn\ cn'' \to cn'\ cn''}$$

Fig. 4 Semantics of core ABS(2)

SYNC- CALL) and (SELF- SYNC- CALL) in that paper, the activated process might be a caller of a synchronous invocation, which has a get operation. To avoid potential deadlock of a wrong *select* implementation, we have prefixed the gets in (COG-SYNC- CALL) and (SELF- SYNC- CALL) with await operations.

The initial configuration of a core ABS program with main function $\{\bar{T} : \bar{x}\ ;\ s\}$ is

$$ob(start, \varepsilon, \{[\mathrm{destiny} \mapsto f_{start}, \bar{x} \mapsto \bot] \mid s\}, \varnothing)$$
$$cog(\mathrm{start}, start)$$

where start and *start* are special cog and object names, respectively, and f_{start} is a fresh future name. As usual, let $\longrightarrow^*$ be the reflexive and transitive closure of $\longrightarrow$.

A configuration *cn* is *sound* if

1. different elements $cog(c, o)$ and $cog(c', o')$ in cn are such that $c \neq c'$ and $o \neq \varepsilon$ implies $o \neq o'$,
2. if $ob(o, a, p, q)$ and $ob(o', a', p', q')$ are different objects in cn such that $a(\text{cog}) = a'(\text{cog})$, then either $p = \text{idle}$ or $p' = \text{idle}$.

We notice that the initial configurations of `core ABS` programs are sound. The following statement guarantees that the property "there is at most one active object per cog" is an invariance of the transition relation.

Proposition 1 *If cn is* sound *and $cn \longrightarrow cn'$, then cn' is sound as well.*

As an example of `core ABS` semantics, in Fig. 7, we have detailed the transitions of the program in Example 2. The non-interested reader may safely skip it.

2.3 Samples of concurrent programs in `core ABS`

The `core ABS` code of two concurrent programs are discussed. These codes will be analyzed in the following sections.

Example 1 Figure 5 collects three different implementations of the factorial function in a 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.

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

Fig. 5 Class `Math`

Yet, `get` does not allow the task to release the cog lock; therefore, the task evaluating `this!fact_g(n-1)` is fated to be delayed forever because its object (and, therefore, the corresponding cog) 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 cog. This means the task of `z!fact_nc(n-1)` may start without waiting for the termination of the caller.

Programs that are particularly hard to verify are those that may manifest misbehaviors according to the scheduler choices. The following example discusses one case.

Example 2 The class `CpxSched` of Fig. 6 defines three methods. Method `m1` asynchronously invokes `m2` on its own argument `y`, passing to it the field `x` as argument. Then, it asynchronously invokes `m2` on the field `x`, passing its same argument `y`. Method `m2` invokes `m3` on the argument `z` and blocks waiting for the result. Method `m3` simply returns.

Next, consider the following main function:

```
{ I x; I y; I z;
  Fut<Fut<Unit>> w ;
  x = new CpxSched(null);
  y = new CpxSched(x);
  z = new cog CpxSched(null);
  w = y!m1(z); }
```

The initial configuration is

$$ob(start, \varepsilon, \{l \mid s\}, \varnothing)cog(\text{start}, start)$$

where $l = [\text{destiny} \mapsto f_{start}, x \mapsto \bot, y \mapsto \bot, z \mapsto$

```
interface I { Fut<Unit> m1(I y); Unit m2(I z);
              Unit m3() ; }

class CpxSched (I u) implements I {
    Fut<Unit> m1(I y) {
        Fut<Unit> h;
        Fut<Unit> g ;
        h = y!m2(u);
        g = u!m2(y);
        return g;
    }
    Unit m2(I z) {
        Fut<Unit> h ;
        h = z!m3();
        h.get;
     }
    Unit m3(){
    }
}
```

Fig. 6 Class `CpxSched`

$$
\begin{array}{l}
\mathit{ob}(\mathit{start}, \varepsilon, \{l \mid s\}, \varnothing)\ \mathit{cog}(\text{start}, \mathit{start}) \\
\quad \longrightarrow^{2} \quad (\textsc{New-Object}) \text{ and } (\textsc{Assign-Local}) \\
\mathit{ob}(\mathit{start}, \varepsilon, \{l[\mathtt{x} \mapsto o] \mid \mathtt{y = new\ C(x);} s''\}, \varnothing)\ \mathit{cog}(\text{start}, \mathit{start})\ \mathit{ob}(o, a_{\mathtt{null}}, \text{idle}, \varnothing) \\
\quad \longrightarrow^{2} \quad (\textsc{New-Object}) \text{ and } (\textsc{Assign-Local}) \\
\mathit{ob}(\mathit{start}, \varepsilon, \{l[\mathtt{x} \mapsto o, \mathtt{y} \mapsto o'] \mid \mathtt{z = new\ cog\ C(null);} s'''\}, \varnothing)\ \mathit{cog}(\text{start}, \mathit{start})\ \mathit{ob}(o, a_{\mathtt{null}}, \text{idle}, \varnothing)\ \mathit{ob}(o', a_o, \text{idle}, \varnothing) \\
\quad \longrightarrow^{2} \quad (\textsc{New-Cog-Object}) \text{ and } (\textsc{Assign-Local}) \\
\mathit{ob}(\mathit{start}, \varepsilon, \{l[\mathtt{x} \mapsto o, \mathtt{y} \mapsto o', \mathtt{z} \mapsto o''] \mid \mathtt{w = y!m1(z);}\}, \varnothing)\ \mathit{cog}(\text{start}, \mathit{start})\ \mathit{ob}(o, a_{\mathtt{null}}, \text{idle}, \varnothing)\ \mathit{ob}(o', a_o, \text{idle}, \varnothing) \\
\qquad \mathit{ob}(o'', [\mathit{cog} \mapsto c, \mathtt{x} \mapsto \mathtt{null}], \text{idle}, \varnothing)\ \mathit{cog}(c, o'') \\
\quad \longrightarrow \quad (\textsc{Async-Call}) \\
\mathit{ob}(\mathit{start}, \varepsilon, \{l[\mathtt{x} \mapsto o, \mathtt{y} \mapsto o', \mathtt{z} \mapsto o''] \mid \mathtt{w = } \text{f;}\}, \varnothing)\ \mathit{cog}(\text{start}, \mathit{start})\ \mathit{ob}(o, a_{\mathtt{null}}, \text{idle}, \varnothing)\ \mathit{ob}(o', a_o, \text{idle}, \varnothing) \\
\qquad \mathit{ob}(o'', [\mathit{cog} \mapsto c, \mathtt{x} \mapsto \mathtt{null}], \text{idle}, \varnothing)\ \mathit{cog}(c, o'')\ \mathit{invoc}(o', f, \mathtt{m1}, o'')\ \mathit{fut}(f, \bot) \\
\quad \longrightarrow \quad (\textsc{Bind-Mtd}) \\
\mathit{ob}(\mathit{start}, \varepsilon, \{l[\mathtt{x} \mapsto o, \mathtt{y} \mapsto o', \mathtt{z} \mapsto o''] \mid \mathtt{w = } \text{f;}\}, \varnothing)\ \mathit{cog}(\text{start}, \mathit{start})\ \mathit{ob}(o, a_{\mathtt{null}}, \text{idle}, \varnothing)\ \mathit{ob}(o', a_o, \text{idle}, \{l_{o'} \mid s_{o'}\}) \\
\mathit{ob}(o'', [\mathit{cog} \mapsto c, \mathtt{x} \mapsto \mathtt{null}], \text{idle}, \varnothing) \qquad \mathit{cog}(c, o'')\ \mathit{fut}(f, \bot) \\
\quad \longrightarrow^{+}\ (\textsc{Activate}) \text{ and twice } (\textsc{Async-Call}) \text{ and } (\textsc{Return}) \\
\mathit{ob}(\mathit{start}, \varepsilon, \{l[\mathtt{x} \mapsto o, \mathtt{y} \mapsto o', \mathtt{z} \mapsto o''] \mid \mathtt{w = } \text{f;}\}, \varnothing)\ \mathit{cog}(\text{start}, \mathit{start})\ \mathit{ob}(o, a_{\mathtt{null}}, \text{idle}, \varnothing)\ \mathit{ob}(o', a_o, \text{idle}, \varnothing) \\
\qquad \mathit{ob}(o'', [\mathit{cog} \mapsto c, \mathtt{x} \mapsto \mathtt{null}], \text{idle}, \varnothing)\ \mathit{cog}(c, o'')\ \mathit{fut}(f, f'')\ \mathit{fut}(f', \bot)\ \mathit{fut}(f'', \bot) \qquad (\star) \\
\qquad \mathit{invoc}(o'', f', \mathtt{m2}, o)\ \mathit{invoc}(o, f'', \mathtt{m2}, o'') \\
\quad \longrightarrow^{2} \quad \text{twice } (\textsc{Bind-Mtd}) \\
\mathit{ob}(\mathit{start}, \varepsilon, \{l[\mathtt{x} \mapsto o, \mathtt{y} \mapsto o', \mathtt{z} \mapsto o''] \mid \mathtt{w = } \text{f;}\}, \varnothing)\ \mathit{cog}(\text{start}, \mathit{start})\ \mathit{ob}(o, a_{\mathtt{null}}, \text{idle}, \{l_o \mid s_o\})\ \mathit{ob}(o', a_o, \text{idle}, \varnothing) \\
\qquad \mathit{ob}(o'', [\mathit{cog} \mapsto c, \mathtt{x} \mapsto \mathtt{null}], \text{idle}, \{l_{o''} \mid s_{o''}\})\ \mathit{cog}(c, o'')\ \mathit{fut}(f, f'')\ \mathit{fut}(f', \bot)\ \mathit{fut}(f'', \bot) \\
\quad \longrightarrow^{+} \quad \text{twice } (\textsc{Activate}) \text{ and twice } (\textsc{Async-Call}) \\
\mathit{ob}(\mathit{start}, \varepsilon, \{l[\mathtt{x} \mapsto o, \mathtt{y} \mapsto o', \mathtt{z} \mapsto o'', \mathtt{w} \mapsto f] \mid \text{idle}\}, \varnothing)\ \mathit{cog}(\text{start}, o)\ \mathit{ob}(o, a_{\mathtt{null}}, \{l_o[\mathtt{h} \mapsto f'''] \mid \mathtt{h.get;} s'_o\}, \varnothing) \\
\qquad \mathit{ob}(o', a_o, \text{idle}, \varnothing)\ \mathit{ob}(o'', [\mathit{cog} \mapsto c, \mathtt{x} \mapsto \mathtt{null}], \{l_{o''}[\mathtt{h} \mapsto f''''] \mid \mathtt{h.get;} s'_{o''}\}, \varnothing)\ \mathit{cog}(c, o'')\ \mathit{fut}(f, f'')\ \mathit{fut}(f', \bot) \\
\qquad \mathit{fut}(f'', \bot)\ \mathit{invoc}(o'', f''', \mathtt{m3}, \varepsilon)\ \mathit{fut}(f', \bot)\ \mathit{fut}(f''', \bot)\ \mathit{invoc}(o, f'''', \mathtt{m3}, \varepsilon)\ \mathit{fut}(f'', \bot)\ \mathit{fut}(f'''', \bot) \\
\quad \longrightarrow^{2} \quad \text{twice } (\textsc{Bind-Mtd}) \\
\mathit{ob}(\mathit{start}, \varepsilon, \{l[\mathtt{x} \mapsto o, \mathtt{y} \mapsto o', \mathtt{z} \mapsto o'', \mathtt{w} \mapsto f] \mid \text{idle}\}, \varnothing)\ \mathit{cog}(\text{start}, o)\ \mathit{ob}(o, a_{\mathtt{null}}, \{l_o[\mathtt{h} \mapsto f'''] \mid \mathtt{h.get;} s'_o\}, \{l'_o \mid \mathtt{skip;}\}) \\
\qquad \mathit{ob}(o', a_o, \text{idle}, \varnothing)\ \mathit{ob}(o'', [\mathit{cog} \mapsto c, \mathtt{x} \mapsto \mathtt{null}], \{l_{o''}[\mathtt{h} \mapsto f''''] \mid \mathtt{h.get;} s'_{o''}\}, \{l'_{o''} \mid \mathtt{skip;}\})\ \mathit{cog}(c, o'') \\
\qquad \mathit{fut}(f, f'')\ \mathit{fut}(f', \bot)\ \mathit{fut}(f'', \bot)\ \mathit{fut}(f', \bot)\ \mathit{fut}(f''', \bot)\ \mathit{fut}(f'', \bot)\ \mathit{fut}(f'''', \bot)
\end{array}
$$

Fig. 7 Reduction of Example 2

$\bot, w \mapsto \bot]$ and s is the statement of the main function. The sequence of transitions of this configuration is illustrated in Fig. 7, where

s', s'',	s''' are the obvious sub-statements of the main function
$l_o =$	$[\text{destiny} \mapsto f'', z \mapsto o'', u \mapsto \bot, h \mapsto \bot]$
$l_{o'} =$	$[\text{destiny} \mapsto f, y \mapsto o'', g \mapsto \bot, h \mapsto \bot]$
$l_{o''} =$	$[\text{destiny} \mapsto f', z \mapsto o, u \mapsto \bot, h \mapsto \bot]$
$l'_o =$	$[\text{destiny} \mapsto f'''']$
$l'_{o''} =$	$[\text{destiny} \mapsto f''']$
$a_{\mathtt{null}} =$	$[\mathit{cog} \mapsto \text{start}, \mathtt{x} \mapsto \mathtt{null}]$
$a_o =$	$[\mathit{cog} \mapsto \text{start}, \mathtt{x} \mapsto o]$
$s_{o'} =$	`h= y!m2(this.x); g= this.x!m2(y); return g;`
$s_o =$	$s_{o''} =$ `h= z!m3(); h.get;`

We notice that the last configuration of Fig. 7 is stuck, i.e., it cannot progress anymore. In fact, it is a deadlock according the forthcoming Definition 1.

2.4 Deadlocks

The definition below identifies deadlocked configurations by detecting chains of dependencies between tasks that cannot progress. To ease the reading, we write

- $p[f.\mathtt{get}]^a$ whenever $p = \{l|s\}$ and s is $x = y.\mathtt{get}; s'$ and $[\![y]\!]_{(a+l)} = f$;
- $p[\mathtt{await}\ f]^a$ whenever $p = \{l|s\}$ and s is $\mathtt{await}\ e?; s'$ and $[\![e]\!]_{(a+l)} = f$;
- $p.f$ whenever $p = \{l|s\}$ and $l(\text{destiny}) = f$.

Definition 1 A configuration *cn* is *deadlocked* if there are

$$\mathit{ob}(o_0, a_0, p_0, q_0), \ldots, \mathit{ob}(o_{n-1}, a_{n-1}, p_{n-1}, q_{n-1}) \in \mathit{cn}$$

and

$$p'_i \in p_i \cup q_i, \qquad \text{with } 0 \le i \le n-1$$

such that (let $+$ be computed modulo n in the following)

1. $p'_0 = p_0[f_0.\mathtt{get}]^{a_0}$ and if $p'_i[f_i.\mathtt{get}]^{a_i}$ then $p'_i = p_i$;
2. if $p'_i[f_i.\mathtt{get}]^{a_i}$ or $p'_i[\mathtt{await}\ f_i]^{a_i}$ then $\mathit{fut}(f_i, \bot) \in \mathit{cn}$ and
   - either $p'_{i+1}[f_{i+1}.\mathtt{get}]^{a_{i+1}}$ and $p'_{i+1} = \{l_{i+1}|s_{i+1}\}$ and $f_i = l_{i+1}(\text{destiny})$;
   - or $p'_{i+1}[\mathtt{await}\ f_{i+1}]^{a_{i+1}}$ and $p'_{i+1} = \{l_{i+1}|s_{i+1}\}$ and $f_i = l_{i+1}(\text{destiny})$;
   - or $p'_{i+1} = p_{i+1} = \text{idle}$ and $a_{i+1}(\mathit{cog}) = a_{i+2}(\mathit{cog})$ and $p'_{i+2}[f_{i+2}.\mathtt{get}]^{a_{i+2}}$ (in this case p_{i+1} is idle, by soundness).

A configuration *cn* is *deadlock-free* if, for every $\mathit{cn} \longrightarrow^{*} \mathit{cn}'$, cn' is not deadlocked. A `core ABS` program is *deadlock-free* if its initial configuration is *deadlock-free*.

According to Definition 1, a configuration is deadlocked when there is a circular dependency between processes. The processes involved in such circularities are performing a `get` or `await` synchronization or they are idle and will never grab the lock because another active process in the same cog will not return. We notice that, by Definition 1, at least one active process is blocked on a `get` synchronization. We also notice that the objects in Definition 1 may be not pairwise different (see Example 1 below). The following examples should make the definition clearer; the reader is recommended to instantiate the definition every time.

1. (Self deadlock)

   $ob(o_1, a_1, \{l_1|x_1 = e_1.\texttt{get}; s_1\}, q_1)$
   $ob(o_2, a_2, \text{idle}, q_2 \cup \{l_2|s_2\})$
   $\mathit{fut}(f_2, \bot)$,

   where $[\![e_1]\!]_{(a_1+l_1)} = l_2(\mathit{destiny}) = f_2$ and $a_1(\mathit{cog}) = a_2(\mathit{cog})$. In this case, the object o_1 keeps the control of its own cog while waiting for the result of a process in o_2. This process cannot be scheduled because the corresponding cog is not released. A similar situation can be obtained with one object:

   $ob(o_1, a_1, \{l_1|x_1 = e_1.\texttt{get}; s_1\}, q_1 \cup \{l_2|s_2\})$
   $\mathit{fut}(f_2, \bot)$,

   where $[\![e_1]\!]_{(a_1+l_1)} = l_2(\mathit{destiny}) = f_2$. In this case, the objects of the Definition 1 are

   $ob(o_1, a_1, p_1, q_1) \quad ob(o_1, a_1, p_2, q_2 \cup \{l_2|s_2\})$

   where $p_1' = p_1 = \{l_1|x_1 = e_1.\texttt{get}; s_1\}$, $p_2' = \{l_2|s_2\}$ and $q_1 = q_2 \cup \{l_2|s_2\}$.

2. (`get`-`await` deadlock)

   $ob(o_1, a_1, \{l_1|x_1 = e_1.\texttt{get}; s_1\}, q_1)$
   $ob(o_2, a_2, \{l_2|\texttt{await}\ e_2?; s_2\}, q_2)$
   $ob(o_3, a_3, \text{idle}, q_3 \cup \{l_3|s_3\})$

   where $l_3(\mathit{destiny}) = [\![e_2]\!]_{a_2+l_2}$, $l_2(\mathit{destiny}) = [\![e_1]\!]_{a_1+l_1}$, $a_1(\mathit{cog}) = a_3(\mathit{cog})$ and $a_1(\mathit{cog}) \neq a_2(\mathit{cog})$. In this case, the objects o_1 and o_2 have different cogs. However, o_2 cannot progress because it is waiting for a result of a process that cannot be scheduled (because it has the same cog of o_1).

3. (`get`-idle deadlock)

   $ob(o_1, a_1, \{l_1|x_1 = e_1.\texttt{get}; s_1\}, q_1)$
   $ob(o_2, a_2, \text{idle}, q_1 \cup \{l_2|s_2\})$
   $ob(o_3, a_3, \{l_3|x_3 = e_3.\texttt{get}; s_3\}, q_3)$
   $ob(o_4, a_4, \text{idle}, q_4 \cup \{l_4|s_4\})$
   $ob(o_5, a_5, \{l_5|x_5 = e_5.\texttt{get}; s_5\}, q_5)$
   $\mathit{fut}(f_1, \bot)$, $\mathit{fut}(f_2, \bot)$, $\mathit{fut}(f_4, \bot)$

   where $f_2 = l_2(\mathit{destiny}) = [\![e_1]\!]_{a_1+l_1}$, $f_4 = l_4(\mathit{destiny}) = [\![e_3]\!]_{a_3+l_3}$, $f_1 = l_1(\mathit{destiny}) = [\![e_5]\!]_{a_5+l_5}$ and $a_2(\mathit{cog}) = a_3(\mathit{cog})$ and $a_4(\mathit{cog}) = a_5(\mathit{cog})$.

A deadlocked configuration has at least one object that is stuck (the one performing the `get` instruction). This means that the configuration may progress, but future configurations will still have one object stuck.

Proposition 2 *If cn is deadlocked and $cn \longrightarrow cn'$, then cn' is deadlocked as well.*

Definition 1 is about runtime entities that have no static counterpart. Therefore, we consider a notion weaker than deadlocked configuration. This last notion will be used in the appendices to demonstrate the correctness of the inference system in Sect. 4.

Definition 2 A configuration cn has

1. A *dependency* (c, c') if

   $ob(o, a, \{l|x = e.\texttt{get}; s\}, q), ob(o', a', p', q') \in cn$

   with $[\![e]\!]_{(a+l)} = f$ and $a(\mathit{cog}) = c$ and $a'(\mathit{cog}) = c'$ and

   (a) either $\mathit{fut}(f, \bot) \in cn$, $l'(\text{destiny}) = f$ and $\{l'|s'\} \in p' \cup q'$;
   (b) or $\mathit{invoc}(o', f, \mathtt{m}, \overline{v}) \in cn$.

2. A *dependency* $(c, c')^{\mathtt{w}}$ if

   $ob(o, a, p, q), ob(o', a', p', q') \in cn$

   and $\{l|\texttt{await}\ e?; s\} \in p \cup q$ and $[\![e]\!]_{(a+l)} = f$ and

   (a) either $\mathit{fut}(f, \bot) \in cn$, $l'(\text{destiny}) = f$ and $\{l'|s'\} \in p' \cup q'$;
   (b) or $\mathit{invoc}(o', f, \mathtt{m}, \overline{v}) \in cn$.

Given a set A of dependencies, let the `get`-*closure* of A, noted $A^{\texttt{get}}$, be the least set such that

1. $A \subseteq A^{\texttt{get}}$;
2. if $(c, c') \in A^{\texttt{get}}$ and $(c', c'')^{[\mathtt{w}]} \in A^{\texttt{get}}$ then $(c, c'') \in A^{\texttt{get}}$, where $(c', c'')^{[\mathtt{w}]}$ denotes either the pair (c', c'') or the pair $(c', c'')^{\mathtt{w}}$.

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

Proposition 3 *If a configuration is deadlocked, then it has a circularity. The converse is false.*

Proof The statement is a straightforward consequence of the definition of deadlocked configuration. To show that the converse is false, consider the configuration

$$ob(o_1, a_1, \{l_1|x_1 = e_1.\texttt{get}; s_1\}, q_1)$$
$$ob(o_2, a_2, \text{idle}, q_2 \cup \{l_2|\texttt{await}\ e_2?; s_2\})$$
$$ob(o_3, a_3, \{l_3|\texttt{return}\ e_3\}, q_3) \quad cn$$

where $l_3(destiny) = [\![e_1]\!]_{a_1+l_1}$, $l_1(destiny) = [\![e_2]\!]_{a_2+l_2}$, $c_2 = a_2(cog) = a_3(cog)$ and $c_1 = a_1(cog) \neq c_2$. This configuration has the dependencies

$$\{(c_1, c_2), (c_2, c_1)^{\text{w}}\}$$

whose `get`-closure contains the circularity (c_1, c_1). However, the configuration is not deadlocked. □

Example 3 The final configuration of Fig. 7 is *deadlocked* according to Definition 1. In particular, there are two objects o and o'' running on different cogs whose active processes have a `get`-synchronization on the result of process in the other object: o is performing a `get` on a future f''' which is $l'_{o''}$(destiny), and o'' is performing a `get` on a future f'''' which is l'_o(destiny) and $\mathit{fut}(f''', \perp)$ and $\mathit{fut}(f'''', \perp)$. We notice that if in the configuration ($\star$), we choose to evaluate $\mathit{invoc}(o'', f', \texttt{m2}, o)$ when the evaluation of $\mathit{invoc}(o, f'', \texttt{m2}, o'')$ has been completed (or conversely) then no deadlock is manifested.

3 Restrictions of `core ABS` in the current release of the contract inference system

The contract inference system we describe in the next section has been prototyped. To verify its feasibility, the current release of the prototype addresses a subset of `core ABS` features. These restrictions permit to ease the initial development of the inference system and do not jeopardize its extension to the full language. Below we discuss the restrictions and, for each of them, either we explain the reasons why they will be retained in the next release(s) or we detail the techniques that will be used to remove them (it is also worth to notice that, notwithstanding the following restrictions, it is still possible to verify large commercial cases, such as a core component of FAS discussed in this paper).

3.1 Returns

`core ABS` syntax admits `return` statements with continuations—see Fig. 1—that, according to the semantics, are executed *after the return value has been delivered to the caller*. These continuations can be hardly controlled by programmers and usually cause unpredictable behaviors, in particular, as regards deadlocks. To increase the precision of our analysis we assume that `core ABS` programs have empty continuations of `return` statements. We observe that this constraint has an exception at run-time: in order to define the semantics of synchronous method invocation, rules (Cog-Sync- Call) and (Self- Sync- Call) append a `cont` f continuation to statements in order to let the right caller be scheduled. Clearly this is the `core ABS` definition of synchronous invocation and it does not cause any misbehavior.

3.2 Fields assignments

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 as follows. If the field is *not of future type*, then we keep field's record structure unchanging. For instance, if a field contains an object of cog 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. It is possible to be more liberal as regards fields assignments. In [19], 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). When the field is *of future type*, then we disallow assignments. In fact, assignments to such fields allow a programmer to define unexpected behaviors. Consider, for example, the following class `Foo` implementing `I_Foo`:

```
class Foo(){
    Fut<T> x ;
    Unit foo_m () {
        Fut<T> local ;
        I_Foo y = new cog Foo() ;
        I_Foo z = new cog Foo() ;
        local = y!foo_n(this) ;
        x = z!foo_n(this) ;
        await local? ;
        await x?
    }
    T foo_n(I_Foo x){ . . . }
}
```

If the main function is

```
    { I_Foo x ;
       Fut<Unit> u ;
       Fut<Unit> v ;
       x = new cog Foo() ;
       u = x!foo_m() ;
       v = x!foo_m() ; }
```

then the two invocations in lines 18 and 19 run in parallel. Each invocation of `foo_m` invokes `foo_n` twice that apparently terminate when `foo_m` returns (with the two final `await` statements). However, this may be not the case because the invocation of `foo_n` line 8 is stored in a field: Consider that the first invocation of `foo_m` (line 18) starts executing, sets the field `x` with its own future f, and then, with the `await` statement in line 9, the second invocation of `foo_m` (line 19) starts. That second invocation *replaces* the content of the field `x` with its own future f': at that point, the second invocation (line 19) will synchronize with f' before terminating, then the first invocation (line 18) will resume and also synchronized with f' before terminating. Hence, even after both invocations (lines 18 and 19) are finished, the invocation of `foo_n` in line 8 may still be running. It is not too difficult to trace such residual behaviors in the inference system [for instance, by grabbing them using a function like $unsync(\Gamma)$]. However, this extension will entangle the inference system and for this reason we decided to deal with generic field assignments in a next release.

It is worth to recall that these restrictions does not apply to local variables of methods, as they can only be accessed by the method in which they are declared. Actually, the foregoing inference algorithm tracks changes of local variables.

3.3 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 inference system, this issue is avoided 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 implementation. In turn, method invocations are defined as the *union* of the possible contracts a method has.

3.4 Synchronization on Booleans

In addition to synchronization on method invocations, `core ABS` permits synchronizations on Booleans, with the statement `await` e. When e 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 e 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 infer, and for this reason, we have restricted the current release of `DF4ABS`.

Nevertheless, the current release of `DF4ABS` adopts a naive solution for `await` statements on Booleans, namely let programmers annotate them 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. In the above case, the object that will modify the Boolean guard is stored in the variable `thread`. Thus, the annotation `[thread]` in the `await` statement. The current inference system of `DF4ABS` is extended with rules dealing with `await` on Boolean guard and, of course, the correctness of the result depends on the correctness of the `await` annotations. A data-flow analysis of Boolean guards in `await` statements may produce a set of cog dependencies that can be used in the inference rule of the corresponding statement. While this is an interesting issue, it will not be our primary concern in the near future.

3.5 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 contract inference system 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. Note that this restriction still allows recursive definition of classes. We will investigate whether it is possible to extend the semi-unification process by associating *regular terms* [7] to objects in the semi-unification process. These regular terms might be derived during the inference by extending the `core ABS` code with annotations, as done for letting synchronizations on Booleans.

3.6 Discussion

The above restrictions do not severely restrict both programming and the precision of our analysis. As we said, despite these limitations, we were able to apply our tool to the industrial-sized case study FAS from SDL Fredhopper and detect that it was deadlock-free. It is also worth to observe that most of our restrictions can be removed with a simple extension of the current implementation. The restriction that may be challenging to remove is the one about recursive object structures, which requires the extension of semi-unification to such structures. We finally observe that other deadlock analysis tools have restrictions similar to those discussed in this section. For instance, `DECO` does not allow futures to be passed around (i.e., futures cannot be returned or put in an object's field) and constraints interfaces to be implemented by at most one class [12]. Therefore, while `DECO` supports the analysis in the presence of field updates, our tool supports futures to be returned.

4 Contracts and the contract inference system

The deadlock detection framework we present in this paper relies on abstract descriptions, called *contracts*, which are extracted from programs by an inference system. The syntax of these descriptions, which is defined in Fig. 8, uses *record names* X, Y, Z, …, and *future names* f, f', …. Future records $\mathbb{r}$, which encode the values of expressions in contracts, may be one of the following:

- a dummy value _ that models primitive types,
- a record name X that represents a placeholder for a value and can be instantiated by substitutions,
- $[cog{:}c, \bar{x}{:}\bar{\mathbb{r}}]$ that defines an object with its cog name c and the values for fields and parameters of the object,
- and $c \rightsquigarrow \mathbb{r}$, which specifies that accessing $\mathbb{r}$ requires control of the cog c (and that the control is to be released once the method has been evaluated). The future record $c \rightsquigarrow \mathbb{r}$ is associated with method invocations: c is the cog of the object on which the method is invoked. The name c in $[cog{:}c, \bar{x}{:}\bar{\mathbb{r}}]$ and $c \rightsquigarrow \mathbb{r}$ will be called *root* of the future record.

Contracts $\mathbb{c}$ collect the method invocations and the dependencies inside statements. In addition to 0, $0.(c, c')$ and $0.(c, c')^{\mathtt{w}}$ that, respectively, represent the empty behavior, the dependencies due to a `get` and an `await` operation, we have basic contracts that deal with method invocations. There are several possibilities:

- $\mathtt{C.m}\,\mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{r}'$ (resp. $\mathtt{C!m}\,\mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{r}'$) specifies that the method `m` of class `C` is going to be invoked *synchronously* (resp. *asynchronously*) on an object $\mathbb{r}$, with arguments $\bar{\mathbb{r}}$, and an object $\mathbb{r}'$ will be returned;
- $\mathtt{C!m}\,\mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{r}'.(c, c')$ indicates that the current method execution requires the termination of method `C!m` running on an object of cog c' in order to release the object of the cog c. This is the contract of an asynchronous method invocation followed by a `get` operation on the same future name.
- $\mathtt{C!m}\,\mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{r}'.(c, c')^{\mathtt{w}}$ indicates that the current method execution requires the termination of method `C.m` running on an object of cog c' in order to progress. This is the contract of an asynchronous method invocation followed by an `await` operation and, possibly but not necessarily, by a `get` operation. In fact, a `get` operation on the same future name does not add any dependency, since it is guaranteed to succeed because of the preceding `await`.

The composite contracts $\mathbb{c} \fatsemi \mathbb{c}'$ and $\mathbb{c} + \mathbb{c}'$ define the abstract behavior of sequential compositions and conditionals, respectively. The contract $\mathbb{c} \parallel \mathbb{c}'$ requires a separate discussion because it models parallelism, which is not explicit in `core ABS` syntax. We will discuss this issue later on.

Example 4 As an example of contracts, let us discuss the terms:

(a) $\mathtt{C.m}[cog{:}c_1, x{:}[cog{:}c_2]]() \to [cog{:}c_1', x{:}[cog{:}c_2]] \fatsemi$
$\mathtt{C.m}[cog{:}c_3, x{:}[cog{:}c_4]]() \to [cog{:}c_3', x{:}[cog{:}c_4]]$;
(b) $\mathtt{C!m}[cog{:}c_1, x{:}[cog{:}c_2]]() \to [cog{:}c_1', x{:}[cog{:}c_2]].(c, c_1) \fatsemi$
$\mathtt{C!m}[cog{:}c_3, x{:}[cog{:}c_4]]() \to [cog{:}c_3', x{:}[cog{:}c_4]].(c, c_3)^{\mathtt{w}}$.

The contract (a) defines a sequence of two synchronous invocations of method `m` of class `C`. We notice that the cog names

$$
\begin{array}{ll}
\mathbb{r} ::= \_ \;\mid\; X \;\mid\; [cog{:}c, \overline{x}{:}\overline{\mathbb{r}}] \;\mid\; c \rightsquigarrow \mathbb{r} & \text{future record} \\
\mathbb{c} ::= 0 \;\mid\; 0.(c, c') \;\mid\; 0.(c, c')^{\mathtt{w}} \;\mid\; \mathtt{C.m}\;\mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}' \;\mid\; \mathtt{C!m}\;\mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}' \;\mid\; \mathtt{C!m}\;\mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}'.(c, c') & \text{contract} \\
\quad \mid\; \mathtt{C!m}\;\mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}'.(c, c')^{\mathtt{w}} \;\mid\; \mathbb{c} \fatsemi \mathbb{c} \;\mid\; \mathbb{c} + \mathbb{c} \;\mid\; \mathbb{c} \parallel \mathbb{c} & \\
\mathbb{x} ::= \mathbb{r} \;\mid\; f & \text{extended future record} \\
\mathbb{z} ::= (\mathbb{r}, \mathbb{c}) \;\mid\; (\mathbb{r}, 0)^{\checkmark} & \text{future reference values}
\end{array}
$$

Fig. 8 Syntax of future records and contracts

c_1' and c_3' are free: This indicates that C.m returns an object of a new cog. As we will see below, a core ABS expression with this contract is `x.m() ; y.m() ;`.

The contract (b) defines an asynchronous invocation of C.m followed by a get statement and an asynchronous one followed by an await. The cog c is one of the callers. A core ABS expression retaining this contract is `u = x!m() ; w = u.get ; v = y!m() ; await v? ;`.

The inference of contracts uses two additional syntactic categories: $\mathbb{x}$ of future record values and $\mathbb{z}$ of typing values. The former one extends future records with *future names*, which are used to carry out the *alias analysis*. In particular, every local variable of methods and every object field and parameter of future type is associated with a future name. Assignments between these terms, such as $x = y$, amounts to copying future names instead of the corresponding values (x and y become aliases). The category $\mathbb{z}$ collects the typing values of future names, which are either $(\mathbb{r}, \mathbb{c})$, for *unsynchronized futures* or $(\mathbb{r}, 0)^{\checkmark}$, for *synchronized ones* (see the comments below).

The abstract behavior of methods is defined by *method contracts* $\mathbb{r}(\bar{\mathbb{s}})\ \{\langle \mathbb{c}, \mathbb{c}' \rangle\}\ \mathbb{r}'$, where $\mathbb{r}$ is the future record of the receiver of the method, $\bar{\mathbb{s}}$ are the future records of the arguments, $\langle \mathbb{c}, \mathbb{c}' \rangle$ is the abstract behavior of the body, where $\mathbb{c}$ is called *synchronized contract* and $\mathbb{c}'$ is called *unsynchronized contract*, and $\mathbb{r}'$ is the future record of the returned object.

Let us explain why method contracts use pairs of contracts. In core ABS, invocations in method bodies are of two types: (*i*) *synchronized*, that is, the asynchronous invocation has a subsequent await or get operation in the method body and (*ii*) *unsynchronized*, the asynchronous invocation has no corresponding await or get in the same method body (Synchronous invocations can be regarded as asynchronous invocations followed by a get).

For example, let

```
x = u!m() ;
await x? ;
y = v!m() ;
```

be the main function of a program (the declarations are omitted). In this statement, the invocation `u!m()` is synchronized before the execution of `v!m()`, which is unsynchronized. core ABS semantics tells us that the body of `u!m()` is *performed before* the body of `v!m()`. However, while this ordering holds for the synchronized part of m, it may not hold for the unsynchronized part. In particular, the unsynchronized part of `u!m()` may run *in parallel* with the body of `v!m()`. For this reason, in this case, our inference system returns the pair

$$\langle \texttt{C!m}\,[cog{:}c'](\,) \to \_.(c, c')^{\mathtt{w}}, \texttt{C!m}\,[cog{:}c''](\,) \to \_\rangle$$

where c, c' and c'' being the cog of the caller, u and v, respectively. Letting $\texttt{C!m}\,[cog{:}c'](\,) \to \_ = \langle \mathbb{c}_u, \mathbb{c}_u' \rangle$ and $\texttt{C!m}\,[cog{:}c''](\,) \to \_ = \langle \mathbb{c}_v, \mathbb{c}_v' \rangle$, one has (see Sects. 5, 6)

$$\langle \texttt{C!m}\,[cog{:}c'](\,) \to \_.(c, c')^{\mathtt{w}}, \texttt{C!m}\,[cog{:}c''](\,) \to \_\rangle$$
$$= \langle \mathbb{c}_u.(c, c')^{\mathtt{w}}, \mathbb{c}_u' \parallel (\mathbb{c}_v \fatsemi \mathbb{c}_v')\rangle$$

that adds the dependency $(c, c')^{\mathtt{w}}$ to the synchronized contract of `u!m()` and makes the parallel (the $\parallel$ operator) of the unsynchronized contract of `u!m()` and the contract of `v!m()`. Of course, in alternative to separating *synchronized* and *unsynchronized contracts*, one might collect all the dependencies in a unique contract. This will imply that the dependencies in different configurations will be gathered in the same set, thus significantly reducing the precision of the analyses in Sects. 5 and 6.

The above discussion also highlights the need of contracts $\mathbb{c} \parallel \mathbb{c}'$. In particular, this operator models *parallel behaviors*, which is not a first class operator in core ABS, while it is central in the semantics (the objects in the configurations). We illustrate the point with a statement similar to the above one, where we have swapped the second and third instruction

```
x = u!m() ;
y = v!m() ;
await x? ;
```

According to core ABS semantics, it is possible that the bodies of `u!m()` and of `v!m()` run in parallel by interleaving their executions. In fact, in this case, our inference system returns the pair of contracts (we keep the same notations as before)

$$\langle\ \texttt{C!m}\,[cog{:}c'](\,) \to \_.(c, c')^{\mathtt{w}} \parallel \texttt{C!m}\,[cog{:}c''](\,) \to \_\,,\ \texttt{C!m}\,[cog{:}c''](\,) \to \_\ \rangle$$

which turns out to be equivalent to

$$\texttt{C!m1}\,[cog{:}c'](\,) \to \_.(c, c')^{\mathtt{w}} \parallel \texttt{C!m2}\,[cog{:}c'](\,) \to \_$$

(see Sects. 5, 6).

The subterm $\mathbb{r}(\bar{\mathbb{s}})$ of the method contract is called *header*; $\mathbb{r}'$ is called *returned future record*. We assume that cog and record names in the header occur linearly. Cog and record names in the header *bind* the cog and record names in $\mathbb{c}$ and in $\mathbb{r}'$. The header and the returned future record, written $\mathbb{r}(\bar{\mathbb{s}}) \to \mathbb{r}'$, are called *contract signature*. In a method contract $\mathbb{r}(\bar{\mathbb{s}})\ \{\langle \mathbb{c}, \mathbb{c}' \rangle\}\ \mathbb{r}'$, cog and record names occurring in $\mathbb{c}$ or $\mathbb{c}'$ or $\mathbb{r}'$ may be *not bound* by header. These *free names* correspond to new cog instructions and will be replaced by fresh cog names during the analysis.

4.1 Inference of contracts

Contracts are extracted from core ABS programs by means of an inference algorithm. Figures 9 and 11 illustrate the set of rules. The following auxiliary operators are used:

expressions and addresses

(T-VAR)
$$\frac{\Gamma(x) = \mathbb{x}}{\Gamma \vdash_c x : \mathbb{x}}$$

(T-FUT)
$$\frac{\Gamma(f) = \mathbb{z}}{\Gamma \vdash_c f : \mathbb{z}}$$

(T-FIELD)
$$\frac{x \notin \mathrm{dom}(\Gamma) \qquad \Gamma(\mathtt{this}.x) = \mathbb{r}}{\Gamma \vdash_c x : \mathbb{r}}$$

(T-VALUE)
$$\frac{\Gamma \vdash_c e : f \qquad \Gamma \vdash_c f : (\mathbb{r}, \mathbb{c})^{[\checkmark]}}{\Gamma \vdash_c e : \mathbb{r}}$$

(T-VAL)
$$\frac{e \quad \textit{primitive value or arithmetic-and-bool-exp}}{\Gamma \vdash_c e : \_}$$

(T-PURE)
$$\frac{\Gamma \vdash_c e : \mathbb{r}}{\Gamma \vdash_c e : \mathbb{r}, 0 \ \triangleright\ \mathtt{true} \mid \Gamma}$$

expressions with side effects

(T-GET)
$$\frac{\Gamma \vdash_c x : f \qquad \Gamma \vdash_c f : (\mathbb{r}, \mathbb{c}) \qquad X, c' \text{ fresh} \qquad \Gamma' = \Gamma[f \mapsto (\mathbb{r}, 0)^{\checkmark}]}{\Gamma \vdash_c x.\mathtt{get} : X, \mathbb{c}.(c, c') \parallel \textit{unsync}(\Gamma') \ \triangleright\ \mathbb{r} = c' \rightsquigarrow X \mid \Gamma'}$$

(T-GET-TICK)
$$\frac{\Gamma \vdash_c x : f \qquad \Gamma \vdash_c f : (\mathbb{r}, \mathbb{c})^{\checkmark} \qquad X, c' \text{ fresh}}{\Gamma \vdash_c x.\mathtt{get} : X, 0 \ \triangleright\ \mathbb{r} = c' \rightsquigarrow X \mid \Gamma}$$

(T-NEWCOG)
$$\frac{\begin{array}{c}\Gamma \vdash_c \overline{e} : \overline{\mathbb{r}} \\ \textit{fields}(\mathtt{C}) = \overline{T\ x} \qquad \textit{param}(\mathtt{C}) = \overline{T'\ x'} \qquad \overline{X}, c' \text{ fresh}\end{array}}{\Gamma \vdash_c \mathtt{new\ cog}\ \mathtt{C}(\overline{e}) : [\textit{cog}{:}c', \overline{x{:}X}, \overline{x'{:}\mathbb{r}}], 0 \ \triangleright\ \mathtt{true} \mid \Gamma}$$

(T-NEW)
$$\frac{\begin{array}{c}\Gamma \vdash_c \overline{e} : \overline{\mathbb{r}} \\ \textit{fields}(\mathtt{C}) = \overline{T\ x} \qquad \textit{param}(\mathtt{C}) = \overline{T'\ x'} \qquad \overline{X} \text{ fresh}\end{array}}{\Gamma \vdash_c \mathtt{new}\ \mathtt{C}(\overline{e}) : [\textit{cog}{:}c, \overline{x{:}X}, \overline{x'{:}\mathbb{r}}], 0 \ \triangleright\ \mathtt{true} \mid \Gamma}$$

(T-AINVK)
$$\frac{\begin{array}{c}\Gamma \vdash_c e : \mathbb{r} \qquad \Gamma \vdash_c \overline{e} : \overline{\mathbb{s}} \\ \textit{class}(\textit{types}(e)) = \mathtt{C} \qquad \textit{fields}(\mathtt{C}) \cup \textit{param}(\mathtt{C}) = \overline{T\ x} \qquad X, \overline{X}, c', f \text{ fresh}\end{array}}{\Gamma \vdash_c e!\mathtt{m}(\overline{e}) : f, 0 \ \triangleright\ [\textit{cog}{:}c', \overline{x{:}X}] = \mathbb{r} \wedge \mathtt{C.m} \preceq \mathbb{r}(\overline{\mathbb{s}}) \to X \mid \Gamma[f \mapsto (c' \rightsquigarrow X,\ \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{s}}) \to X)]}$$

(T-SINVK)
$$\frac{\begin{array}{c}\Gamma \vdash_c e : \mathbb{r} \qquad \Gamma \vdash_c \overline{e} : \overline{\mathbb{s}} \\ \textit{class}(\textit{types}(e)) = \mathtt{C} \qquad \textit{fields}(\mathtt{C}) \cup \textit{param}(\mathtt{C}) = \overline{T\ x} \qquad X, \overline{X} \text{ fresh}\end{array}}{\Gamma \vdash_c e.\mathtt{m}(\overline{e}) : X, \mathtt{C.m}\ \mathbb{r}(\overline{\mathbb{s}}) \to X \parallel \textit{unsync}(\Gamma) \ \triangleright\ [\textit{cog}{:}c', \overline{x{:}X}] = \mathbb{r} \wedge \mathtt{C.m} \preceq \mathbb{r}(\overline{\mathbb{s}}) \to X \mid \Gamma}$$

Fig. 9 Contract inference for expressions and expressions with side effects

- *fields*(`C`) and *param*(`C`), respectively, return the sequence of fields and parameters and their types of a class `C`. Sometimes we write $\textit{fields}(\mathtt{C}) = \overline{T\ x},\ \overline{\mathtt{Fut<}T'\mathtt{>}\ x'}$ to separate fields with a non-future type by those with future types. Similarly for parameters;
- *types*(e) returns the type of an expression e, which is either an interface (when e is an object) or a data type;
- *class*(I) returns the unique (see the restriction *Interfaces* in Sect. 3) class implementing I; and
- *mname*($\bar{M}$) returns the sequence of method names in the sequence $\bar{M}$ of method declarations.

The inference algorithm uses constraints $\mathcal{U}$, which are defined by the following syntax

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

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 *semi-unification* constraint; the constraint $\mathcal{U} \wedge \mathcal{U}'$ is the conjunction of $\mathcal{U}$ and $\mathcal{U}'$. We use *semi-unification* constraints [20] to deal with method invocations: basically, in $\mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{s} \preceq \mathbb{r}'(\overline{\mathbb{r}}') \to \mathbb{s}'$, the left hand side of the constraint corresponds to the method's formal parameter, $\mathbb{r}$ being the record of `this`, $\overline{\mathbb{r}}$ being the records of the parameters and $\mathbb{r}'$ being the record of the returned value, while the right hand side corresponds to the actual parameters of the call, and the actual returned value. The meaning of this constraint is that the actual parameters and returned value must match the specification given by the formal parameters, like in a standard unification: The necessity of semi-unification appears when we call several times the same method. Indeed, there, unification would require that the actual parameters of the different calls must all have the same records, while with semi-unification all method calls are managed independently.

The judgments of the inference algorithm have a typing context Γ mapping variables to extended future records, future names to future name values and methods to their signatures. They have the following form:

- $\Gamma \vdash_c e : \mathbb{x}$ for pure expressions e and $\Gamma \vdash_c f : \mathbb{z}$ for future names f, where c is the cog name of the object executing the expression and $\mathbb{x}$ and $\mathbb{z}$ are their inferred values.
- $\Gamma \vdash_c z : \mathbb{r}, \mathbb{c} \triangleright \mathcal{U} \mid \Gamma'$ for expressions with side effects z, where c, and $\mathbb{x}$ are as for pure expressions e, $\mathbb{c}$ is the

contract for z created by the inference rules, $\mathcal{U}$ is the generated constraint, and Γ' is the environment Γ *with updates* of variables and future names. We use the same judgment for pure expressions; in this case $\mathbb{c} = 0, \mathcal{U} = \texttt{true}$ and $\Gamma' = \Gamma$.

– for statements s: $\Gamma \vdash_c s : \mathbb{c} \rhd \mathcal{U} \mid \Gamma'$ where c, $\mathbb{c}$ and $\mathcal{U}$ are as before, and Γ' is the environment obtained after the execution of the statement. The environment may change because of variable updates.

Since Γ is a function, we use the standard predicates $x \in \mathrm{dom}(\Gamma)$ or $x \notin \mathrm{dom}(\Gamma)$. Moreover, given a function Γ, we define $\Gamma[x \mapsto \mathbb{x}]$ to be the following function

$$\Gamma[x \mapsto \mathbb{x}](y) = \begin{cases} \mathbb{x} & \text{if } y = x \\ \Gamma(y) & \text{otherwise} \end{cases}$$

We also let $\Gamma|_{\{x_1,\ldots,x_n\}}$ be the function

$$\Gamma|_{\{x_1,\ldots,x_n\}}(y) = \begin{cases} \Gamma(y) & \text{if } y \in \{x_1,\ldots,x_n\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Moreover, provided that $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma') = \emptyset$, the environment $\Gamma + \Gamma'$ be defined as follows

$$(\Gamma + \Gamma')(x) \overset{def}{=} \begin{cases} \Gamma(x) & \text{if } x \in \mathrm{dom}(\Gamma) \\ \Gamma'(x) & \text{if } x \in \mathrm{dom}(\Gamma') \end{cases}$$

Finally, we write $\Gamma(\texttt{this}.x) = \mathbb{x}$ whenever $\Gamma(\texttt{this}) = [cog{:}c, x : \mathbb{x}, \bar{x} : \bar{\mathbb{x}}']$ and we let

$$\texttt{Fut}(\Gamma) \overset{def}{=} \{x \mid \Gamma(x) \text{ is a future name}\}$$

$$unsync(\Gamma) \overset{def}{=} \mathbb{c}_1 \parallel \cdots \parallel \mathbb{c}_n$$

where $\{\mathbb{c}_1, \ldots, \mathbb{c}_n\} = \{\mathbb{c}' \mid \text{there are } f, \mathbb{r} : \Gamma(f) = (\mathbb{r}, \mathbb{c}')\}$.

The inference rules for expressions and future names are reported in Fig. 9. They are straightforward, except for (T-Value) that performs the dereference of variables and return the future record stored in the future name of the variable. (T-Pure) lifts the judgment of a pure expression to a judgment similar to those for expressions with side effects. This expedient allows us to simplify rules for statements.

Figure 9 also reports inference rules for expressions with side effects. Rule (T-Get) deals with the $x.\texttt{get}$ synchronization primitive and returns the contract $\mathbb{c}.(c, c') \parallel unsync(\Gamma)$, where $\mathbb{c}$ is stored in the future name of x and (c, c') represents a dependency between the cog of the object executing the expression and the root of the expression. The constraint $\mathbb{r} = c' \rightsquigarrow X$ is used to extract the root c' of $\mathbb{r}$. The contract $\mathbb{c}$ may have two shapes: either (1) $\mathbb{c} = \texttt{C!m}\,\mathbb{r}(\bar{\mathbb{s}}) \to \mathbb{r}'$ or (2) $\mathbb{c} = 0$. The subterm $unsync(\Gamma)$ lets us collect all the contracts in Γ *that are stored in future names that are not checkmarked*. In fact, these contracts correspond to previous asynchronous invocations without any corresponding synchronization (`get` or `await` operation) in the body. The evaluations of these invocations may interleave with the evaluation of the expression $x.\texttt{get}$. For this reason, the intended meaning of $unsync(\Gamma)$ is that the dependencies generated by the invocations must be collected *together* with those generated by $\mathbb{c}.(c, c')$. We also observe that the rule updates the environment by checkmarking the value of the future name of x and by replacing the contract with 0 (because the synchronization has been already performed). This allows subsequent `get` (and `await`) operations on the same future name not to modify the contract (in fact, in this case, they are operationally equivalent to the `skip` statement)—see (T-Get-Tick).

Rule (T-NewCog) returns a record with a new cog name. This is in contrast with (T-New), where the cog of the returned record is the *same* of the object executing the expression.[2]

Rule (T-AInvk) derives contracts for asynchronous invocations. Since the dependencies created by these invocations influence the dependencies of the synchronized contract only if a subsequent `get` or `await` operation is performed, the rule stores the invocation into a fresh future name of the environment and returns the contract 0. This models `core ABS` semantics that lets asynchronous invocations be synchronized by explicitly getting or awaiting on the corresponding future variable, see rules (T-Get) and (T-Await). The future name storing the invocation is returned by the judgment. On the contrary, in rule (T-SInvk), which deals with synchronous invocations, the judgment returns a contract that is the invocation (because the corresponding dependencies must be added to the current ones) in parallel with the unsynchronized asynchronous invocations stored in Γ.

The inference rules for statements are collected in Fig. 10. The first three rules define the inference of contracts for assignment. There are two types of assignments: those updating fields and parameters of the `this` object and the other ones. For every type, we need to address the cases of updates with values that are expressions (with side effects) (rules (T-Field-Record) and (T-Var-Record)), or future names (rule (T-Var-Future)). Rules for fields and parameters updates enforce that their future records are unchanging, as discussed in Sect. 3. Rule (T-Var-Future) defines the management

[2] 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 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 (T-New) and (T-NewCog), acting as if `init` and `run` are always absent.

statements

$$\text{T-Skip}\quad \Gamma \vdash_c \texttt{skip} : 0 \rhd \texttt{true} \mid \Gamma$$

$$\text{(T-Field-Record)}\quad \dfrac{x \notin \mathrm{dom}(\Gamma) \qquad \Gamma(\texttt{this}.x) = \mathbb{r} \qquad \Gamma \vdash_c z : \mathbb{r}', \mathbb{c} \rhd \mathcal{U} \mid \Gamma'}{\Gamma \vdash_c x = z : \mathbb{c} \rhd \mathcal{U} \wedge \mathbb{r} = \mathbb{r}' \mid \Gamma'}$$

$$\text{(T-Var-Record)}\quad \dfrac{\Gamma(x) = \mathbb{r} \qquad \Gamma \vdash_c z : \mathbb{r}', \mathbb{c} \rhd \mathcal{U} \mid \Gamma'}{\Gamma \vdash_c x = z : \mathbb{c} \rhd \mathcal{U} \mid \Gamma'[x \mapsto \mathbb{r}']}$$

$$\text{(T-Var-Future)}\quad \dfrac{\Gamma(x) = f \qquad \Gamma \vdash_c z : f', \mathbb{c} \rhd \mathcal{U} \mid \Gamma'}{\Gamma \vdash_c x = z : \mathbb{c} \rhd \mathcal{U} \mid \Gamma'[x \mapsto f']}$$

$$\text{(T-Await)}\quad \dfrac{\Gamma \vdash_c e : f \qquad \Gamma \vdash_c f : (\mathbb{r}, \mathbb{c}) \qquad X, c' \text{ fresh} \qquad \Gamma' = \Gamma[f \mapsto (\mathbb{r}, 0)^{\checkmark}]}{\Gamma \vdash_c \texttt{await } e? : \mathbb{c}.(c, c')^{\mathtt{w}} \parallel \mathit{unsync}(\Gamma') \rhd \mathbb{r} = c' \rightsquigarrow X \mid \Gamma'}$$

$$\text{(T-Await-Tick)}\quad \dfrac{\Gamma \vdash_c e : f \qquad \Gamma \vdash_c f : (\mathbb{r}, \mathbb{c})^{\checkmark} \qquad X, c' \text{ fresh}}{\Gamma \vdash_c \texttt{await } e? : 0 \rhd \mathbb{r} = c' \rightsquigarrow X \mid \Gamma}$$

$$\text{(T-If)}\quad \dfrac{\begin{array}{c}\Gamma \vdash_c e : \texttt{Bool} \qquad \Gamma \vdash_c s_1 : \mathbb{c}_1 \rhd \mathcal{U}_1 \mid \Gamma_1 \qquad \Gamma \vdash_c s_2 : \mathbb{c}_2 \rhd \mathcal{U}_2 \mid \Gamma_2 \\ \mathcal{U} = \Big(\bigwedge_{x \in \mathrm{dom}(\Gamma)} \Gamma_1(x) = \Gamma_2(x)\Big) \wedge \Big(\bigwedge_{x \in \texttt{Fut}(\Gamma)} \Gamma_1(\Gamma_1(x)) = \Gamma_2(\Gamma_2(x))\Big) \qquad \Gamma' = \Gamma_1 + \Gamma_2|_{\{f \mid f \notin \Gamma_2(\texttt{Fut}(\Gamma))\}}\end{array}}{\Gamma \vdash_c \texttt{if } e \; \{\, s_1 \,\} \texttt{ else } \{\, s_2 \,\} : \mathbb{c}_1 + \mathbb{c}_2 \rhd \mathcal{U}_1 \wedge \mathcal{U}_2 \wedge \mathcal{U} \mid \Gamma'}$$

$$\text{(T-Seq)}\quad \dfrac{\Gamma \vdash_c s_1 : \mathbb{c}_1 \rhd \mathcal{U}_1 \mid \Gamma_1 \qquad \Gamma_1 \vdash_c s_2 : \mathbb{c}_2 \rhd \mathcal{U}_2 \mid \Gamma_2}{\Gamma \vdash_c s_1; s_2 : \mathbb{c}_1 \mathbin{⨟} \mathbb{c}_2 \rhd \mathcal{U}_1 \wedge \mathcal{U}_2 \mid \Gamma_2}$$

$$\text{(T-Return)}\quad \dfrac{\Gamma \vdash_c e : \mathbb{r} \qquad \Gamma(\texttt{destiny}) = \mathbb{r}'}{\Gamma \vdash_c \texttt{return } e : 0 \rhd \mathbb{r} = \mathbb{r}' \mid \Gamma}$$

Fig. 10 Contract inference for statements

$$\text{(T-Method)}\quad \dfrac{\begin{array}{c}\mathit{fields}(\texttt{C}) \cup \mathit{param}(\texttt{C}) = \overline{T_f \; x} \; \overline{\texttt{Fut<}T_f'\texttt{>} \; x'} \qquad c, \overline{X}, \overline{X'}, \overline{Y}, \overline{Y'}, \overline{W}, \overline{W'}, \overline{f}, \overline{f'}, \overline{f''}, Z \; \mathit{fresh} \\ \Gamma' = \overline{y{:}Y} + \overline{y'{:}f'} + \overline{w{:}W} + \overline{w'{:}f''} + \overline{f{:}(\overline{X'}, 0)} + \overline{f'{:}(\overline{Y'}, 0)} + \overline{f''{:}(\overline{W'}, 0)} \\ \Gamma + \texttt{this} : [\mathit{cog}{:}c,\; \overline{x{:}X}, \overline{x{:}f}] + \Gamma' + \texttt{destiny} : Z \vdash_c s : \mathbb{c} \rhd \mathcal{U} \mid \Gamma'' \qquad \overline{T},\; \overline{T_f},\; \overline{T_l} \text{ are not future types}\end{array}}{\texttt{C}, \Gamma \vdash T \; \texttt{m} \; (\overline{T \; y}, \overline{\texttt{Fut<}T'\texttt{>} \; y'})\{\overline{T_l \; w}; \; \overline{\texttt{Fut<}T_l'\texttt{>} \; w'}; \; s\} \; : \; \begin{array}{c}[\mathit{cog} : c, \overline{x{:}X},\; \overline{x'{:}X'}](\overline{Y}, \overline{Y'})\{\langle \mathbb{c}, \mathit{unsync}(\Gamma'')\rangle\} \; Z \\ \rhd \mathcal{U} \wedge [\mathit{cog} : c, \overline{x{:}X},\; \overline{x'{:}X'}](\overline{Y}, \overline{Y'}) \rightarrow Z = \texttt{C.m}\end{array}}$$

$$\text{(T-Class)}\quad \dfrac{\texttt{C}, \Gamma \vdash \overline{M} : \overline{\mathbb{C}} \rhd \overline{\mathcal{U}}}{\Gamma \vdash \texttt{class C}(\overline{T \; x}) \; \{\overline{T' \; x'}; \quad \overline{M}\} : \overline{\texttt{C}.\mathit{mname}(M) \mapsto \mathbb{C}} \rhd \overline{\mathcal{U}}}$$

$$\text{(T-Program)}\quad \dfrac{\Gamma \vdash \overline{C} : \overline{S} \rhd \overline{\mathcal{U}} \qquad \overline{X}, \overline{X'}, \overline{f} \; \mathit{fresh} \qquad \Gamma + \overline{x{:}X} + \overline{x'{:}f} + \overline{f{:}(X', 0)} \vdash_{\text{start}} s : \mathbb{c} \rhd \mathcal{U} \mid \Gamma' \qquad \overline{T} \text{ are not future types}}{\Gamma \vdash \overline{I} \; \overline{C} \; \{\overline{T \; x}; \; \overline{\texttt{Fut<}T'\texttt{>} \; x'}; \; s\} : \overline{S}, \langle \mathbb{c}, \mathit{unsync}(\Gamma')\rangle \rhd \mathcal{U} \wedge \overline{\mathcal{U}}}$$

Fig. 11 Contract rules of method and class declarations and programs

of aliases: Future variables are always updated with future names and never with future names' values.

Rules (T- Await) and (T- AwaitTick) deal with the `await` synchronization when applied to a simple future lookup x?. They are similar to the rules (T- Get) and (T- Get- Tick).

Rule (T- If) defines contracts for conditionals. In this case, we collect the contracts $\mathbb{c}_1$ and $\mathbb{c}_2$ of the two branches, with the intended meaning that the dependencies defined by $\mathbb{c}_1$ and $\mathbb{c}_2$ are always kept separated. As regards the environments, the rule constraints the two environments Γ_1 and Γ_2 produced by typing of the two branches to *be the same* on variables in $\mathrm{dom}(\Gamma)$ *and* on the values of future names bound to variables in $\texttt{Fut}(\Gamma)$. However, the two branches may have different unsynchronized invocations that are not bound to any variable. The environment $\Gamma_1 + \Gamma_2|_{\{f \mid f \notin \Gamma_2(\texttt{Fut}(\Gamma))\}}$ allows us to collect all them.

Rule (T- Seq) defines the sequential composition of contracts. Rule (Return) constrains the record of `destiny`, which is an identifier introduced by (T- Method), shown in Fig. 11, for storing the return record.

The rules for method and class declarations are defined in Fig. 11. Rule (T- Method) derives the method contract of T m $(\overline{T\ x})\{\overline{T'\ u}; s\}$ by typing s in an environment extended with `this`, `destiny` [that will be set by `return` statements, see (T- Return)], the arguments $\bar{x}$, and the local variables $\bar{u}$. In order to deal with alias analysis of future variables, we separate fields, parameters, arguments and local variables with future types from the other ones. In particular, we associate future names with the former ones and bind future names to record variables. As discussed above, the abstract behavior of the method body is a pair of contracts, which is $\langle \mathbb{c}, \mathit{unsync}(\Gamma'') \rangle$ for (T- METHOD). This term $\mathit{unsync}(\Gamma'')$ collects all the contracts in Γ'' *that are stored in future names that are not checkmarked*. In fact, these contracts correspond to asynchronous invocations without any synchronization (`get` or `await` operation) in the body. These invocations will be evaluated *after* the termination of the body—they are the *unsynchronized contract*.

The rule (T- 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 analyzers in Sects. 5 and 6. The rule (T- PROGRAM) derives the contract of a `core ABS` program by typing the main function in the same way as it was a body of a method.

The contract class tables of the classes in a program derived by the rule (T- CLASS) will be noted CCT. We will address the contract of `m` of class `C` by CCT(`C.m`). In the following, we assume that every `core ABS` program is a triple (CT, $\{\overline{T\ x\ ;}\ s\}$, CCT), where CT is the class table, $\{\overline{T\ x\ ;}\ s\}$ is the main function, and CCT is its contract class table. By rule (PROGRAM), analyzing (the deadlock freedom of) a program amounts to verifying the contract of the main function with a record for `this` which associates a special cog name called start with the *cog* field (start is intended to be the cog name of the object *start*).

Example 5 The methods of `Math` in Fig. 5 have the following contracts, once the constraints are solved (we always simplify $\mathbb{c} \fatsemi 0$ into $\mathbb{c}$):

– `fact_g` has contract

$$[cog{:}c](\_)\ \{\langle 0 + \texttt{Math!fact\_g}\ [cog{:}c](\_) \to \_.(c, c), 0 \rangle\}\ \_.$$

The name c in the header refers to the cog name associated with `this` in the code and binds the occurrences of c in the body. The contract body has a recursive invocation to `fact_g`, which is performed on an object in the same cog c and followed by a `get` operation. This operation introduces a dependency (c, c). 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 cog as `this`.

– `fact_ag` has contract

$$[cog{:}c](\_)\ \{\langle 0 + \texttt{Math!fact\_ag}\ [cog{:}c](\_) \to \_.(c, c)^{\mathrm{w}}, 0 \rangle\}\ \_.$$

In this case, the presence of an `await` statement in the method body produces a dependency $(c, c)^{\mathrm{w}}$. The subsequent `get` operation does not introduce any dependency because the future name has a check-marked value in the environment. In fact, in this case, the success of `get` is guaranteed, provided the success of the `await` synchronization.

– `fact_nc` has contract

$$[cog{:}c](\_)\ \{\langle 0 + \texttt{Math!fact\_nc}\ [cog{:}c'](\_) \to \_.(c, c'), 0 \rangle\}\ \_.$$

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 c in the header). This because the recursive invocation is performed on an object of a new cog (which is therefore different from c). As a consequence, the dependency added by the `get` relates the cog c of `this` with the new cog c'.

Example 6 Figure 12 displays the contracts of the methods of class `CpxSched` in Fig. 6.

According to the contract of the main function, the two invocations of `m2` are the second arguments of $\|$ operators. This will give rise, in the analysis of contracts, to the union of the corresponding cog relations.

– method `m1` has contract

$$[cog{:}c, \texttt{x} : [cog{:}c', \texttt{x} : X]]([cog{:}c'', \texttt{x} : Y])\ \{\langle 0, \mathbb{c} \rangle\}\ c' \rightsquigarrow \_.$$

where $\mathbb{c} = \texttt{CpxSched!m2}\ [cog{:}c'', \texttt{x} : Y]([cog{:}c', \texttt{x} : X]) \to c'' \rightsquigarrow \_ \ \|\ \texttt{CpxSched!m2}\ [cog{:}c', \texttt{x} : X]([cog{:}c'', \texttt{x} : Y]) \to c' \rightsquigarrow \_$

– method `m2` has contract

$$[cog{:}c, \texttt{x} : X]([cog{:}c', \texttt{x} : Y])\ \{\langle \texttt{CpxSched!m3}\ [cog{:}c', \texttt{x} : Y](\_) \to \_.(c, c'), 0 \rangle\}\ \_.$$

– method `m3` has contract

$$[cog{:}c, \texttt{x} : X](\,)\ \{\langle 0, 0 \rangle\}\ \_.$$

Fig. 12 Contracts of `CpxSched`

We notice that the inference system of contracts discussed in this section is modular because, when programs are organized in different modules, it partially supports the separate contract inference of modules with a well-founded ordering relation (for example, if there are two modules, classes in the second module use definitions or methods in the first one, but not conversely). In this case, if a module `B` includes a module `A`, then a patch to a class of `B` amounts to inferring contracts for `B` only. On the contrary, a patch to a class of `A` may also require a new contract inference of `B`.

4.2 Correctness results

In our system, the ill-typed programs are those manifesting a failure of the semiunification process, which does not address misbehaviors. In particular, a program may be well-typed and still manifest a deadlock. In fact, in systems with *behavioral types*, one usually demonstrates that

1. in a well-typed program, every configuration *cn* has a behavioral type, let us call it $\mathtt{bt}(cn)$;
2. if $cn \rightarrow cn'$, then there is a relationship between $\mathtt{bt}(cn)$ and $\mathtt{bt}(cn')$;
3. the relationship in 2 preserves a given property (in our case, deadlock freedom).

Item 1, in the context of the inference system of this section, means that the program has a contract class table. Its proof needs a contract system for configurations, which we have defined in "Appendix 1". The theorem corresponding to this item is Theorem 3.

Item 2 requires the definition of a relation between contracts, called *later-stage relation* in "Appendix 1". This later-stage relation is a syntactic relationship between contracts whose basic law is that a method invocation is larger than the instantiation of its method contract (the other laws, except $0 \trianglelefteq \mathbb{c}$ and $\mathbb{c}_i \trianglelefteq \mathbb{c}_1 + \mathbb{c}_2$, are congruence laws).

The statement that relates the later-stage relationship to `core ABS` reduction is Theorem 4. It is worth to observe that all the theoretical development upto this point are useless if the later-stage relation conveyed no relevant property. This is the purpose of item 3, which requires the definition of *contract models* and the proof that deadlock freedom is preserved by the models of contracts in later-stage relation. The reader can find the proofs of these statements in the "Appendices 2 and 3" (they correspond to the two analysis techniques that we study).

5 The fixpoint analysis of contracts

The first algorithm we define to analyze contracts uses the standard Knaster–Tarski fixpoint technique. We first give an informal introduction of the notion used in the analysis and start to formally define our algorithm in Sect. 5.1 (a simplified version of the algorithm may be found in [16], see also Sect. 8).

Based on a contract class table and a main contract (both produced by the inference system in Sect. 4), our fixpoint algorithm generates models that encode the dependencies between cogs that may occur during the program's execution. These models, called *lams* (an acronym for deadLock Analysis Models [17,18]), are sets of relations between cog names, each relation representing a possible configuration of program's execution. Consider for instance the main function:

```
{
  I x ; I y ; Fut<Unit> f ;
  x = new cog C() ;
  y = new cog C() ;
  f = x!m() ;
  await f? ;
  f = y!m() ;
  await f? ;
}
```

In this case, the configurations of the program may be represented by two relations: one containing a dependency between the cog name *start* and the cog name of *x* and the other containing a dependency between *start* and the cog name of *y*. This would be represented by the following lam (where c_x and c_y, respectively, being the cog names of *x* and *y*):

$$[\,(c, c_x)^{\mathtt{w}}\,],\ \ [\,(c, c_y)^{\mathtt{w}}\,]$$

(in order to ease the parsing of the formula, we are representing relations with the notation $[\,\cdot\,]$, and we have removed the outermost curly brackets). Our algorithm, being a fixpoint analysis, returns *the* lam of a program by computing a sequence of approximants. In particular, the algorithm performs the following steps:

1. compute a new approximant of the lam of every method using the abstract class table and the previously computed lams;
2. reiterate step 1 till a fixed approximant—say *n* (if a fixpoint is found before, go to 4);
3. when the *n*-th approximant is computed then *saturate*, i.e., compute the next approximants by reusing the same cog names (then a fixpoint is eventually found);
4. replace the method invocations in the main contract with the corresponding values; and
5. analyze the result of 4 by looking for a circular dependency in one of the relations of the computed lam (in such case, a possible deadlock is detected).

The critical issue of our algorithm is the creation of fresh cog names at each step 1, because of free names in method contracts (that correspond to new cogs created during method's execution). For example, consider the contract of `Math.fact_nc` that has been derived in Sect. 4

$$\begin{array}{l}\texttt{Math.fact\_nc}\,[cog{:}c](\_)\\ \quad \{\langle 0+\texttt{Math!fact\_nc}\,[cog{:}c'](\_) \to \_.(c,c'), 0\rangle\}\_\end{array}$$

According to our definitions, the cog name c' is free. In this case, our fixpoint algorithm will produce the following sequence of lams when computing the model of $\texttt{Math.fact\_nc}[cog{:}c_0](\_)$:

$$\begin{array}{ll}
\textit{approximant 0}: & \langle\, [\, \varnothing\,]\, , 0\rangle\\
\textit{approximant 1}: & \langle\, [\, (c_0,c_1)\,]\, , 0\rangle\\
\textit{approximant 2}: & \langle\, [\, (c_0,c_1),(c_1,c_2)\,]\, , 0\rangle\\
\ldots & \\
\textit{approximant n}: & \langle\, [\, (c_0,c_1),(c_1,c_2),\ldots(c_{n-1},c_n)\,]\, , 0\rangle\\
\ldots &
\end{array}$$

While every lam in the above sequence is strictly larger than the previous one, an upper bound element cannot be obtained by iterating the process. Technically, the lam model *is not a complete partial order* (the ascending chains of lams may have infinite length and no upper bound).

In order to circumvent this issue and to get a decision on deadlock freedom in a finite number of steps, we use a *saturation argument*. If the n-th approximant is not a fixpoint, then the $(n+1)$-th approximant is computed by *reusing the same cog names used by the n-th approximant* (no additional cog 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, for $\texttt{Math.fact\_nc}[cog{:}c_0](\_)$, the n-th approximant returns the pairs of lams

$$\langle\, [\, (c_0,c_1),\ldots,(c_{n-1},c_n)\,]\, , 0\rangle.$$

Saturating at this stage yields the lam

$$\langle\, [\, (c_0,c_1),\ldots,(c_{n-1},c_n),(c_1,c_1)\,]\, , 0\rangle$$

that contains a circular dependency—the pair (c_1,c_1)—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.

Note finally that a lam is the result of the analysis of one contract. Hence, to match the structures that are generated during the type inference, our analysis uses three extensions of lams: (1) a pair of lams $\langle \mathcal{L}, \mathcal{L}'\rangle$ for analyzing pairs of contracts $\langle \mathbb{c}, \mathbb{c}'\rangle$; (2) *parameterized* pair of lams $\lambda\overline{c}.\langle \mathcal{L}, \mathcal{L}'\rangle$ for analyzing methods: Here, $\overline{c}$ are the cog names in the header of the method (the `this` object and the formal parameters), and $\langle \mathcal{L}, \mathcal{L}'\rangle$ is the result of the analysis of the contract pair typing the method; and (3) lam tables $(\ldots, \lambda\overline{c_i}.\langle \mathcal{L}_i, \mathcal{L}'_i\rangle, \ldots)$ that maps each method in the program to its current approximant. We observe that $\lambda\overline{c}.\langle \mathcal{L}, \mathcal{L}'\rangle$ is $\langle \mathcal{L}, \mathcal{L}'\rangle$ whenever $\overline{c}$ is empty.

5.1 Lams and lam operations

The following definition formally introduce the notion of lam.

Definition 3 A relation on cog names is a set of pairs either of the form (c_1,c_2) or of the form $(c_1,c_2)^{\mathrm{w}}$, generically represented as $(c_1,c_2)^{[\mathrm{w}]}$. We denote such relation by $[\,(c_{i_0},c_{i_1})^{[\mathrm{w}]},\ldots,(c_{i_{n-1}},c_{i_n})^{[\mathrm{w}]}\,]$.

A *lam*, ranged over $\mathcal{L}, \mathcal{L}', \ldots$, is a set of relations on cog names. Let 0 be the lam $[\,\varnothing\,]$ and let $\textit{cog\_names}(\mathcal{L})$ be the cog names occurring in $\mathcal{L}$.

The pre-order relation between lam, pair of lams and parameterized pair of lam, noted $\Subset$ is defined below. This pre-order is central to prove that our algorithm indeed computes a fixpoint.

Definition 4 Let $\mathcal{L}$ and $\mathcal{L}'$ be lams and κ be an injective function between cog names. We note $\mathcal{L} \Subset_\kappa \mathcal{L}'$ iff for every $L \in \mathcal{L}$ there is $L' \in \mathcal{L}'$ with $\kappa(L) \subseteq L'$. Let

– $\lambda\overline{c}.\langle \mathcal{L}_1, \mathcal{L}'_1\rangle \Subset_\kappa \lambda\overline{c}.\langle \mathcal{L}_2, \mathcal{L}'_2\rangle$ iff κ is the identity on $\overline{c}$ and $\langle \mathcal{L}_1, \mathcal{L}'_1\rangle \Subset_\kappa \langle \mathcal{L}_2, \mathcal{L}'_2\rangle$.

Let also $\Subset$ be the relation

– $\mathcal{L} \Subset \mathcal{L}'$ iff there is κ such that $\mathcal{L} \Subset_\kappa \mathcal{L}'$;
– $\lambda\overline{c}.\langle \mathcal{L}_1, \mathcal{L}'_1\rangle \Subset \lambda\overline{c}.\langle \mathcal{L}_2, \mathcal{L}'_2\rangle$ iff there is κ such that $\lambda\overline{c}.\langle \mathcal{L}_1, \mathcal{L}'_1\rangle \Subset_\kappa \lambda\overline{c}.\langle \mathcal{L}_2, \mathcal{L}'_2\rangle$.

The set of lams with the $\Subset$ relation is a pre-order with a bottom element, which is either 0 or $\lambda\overline{c}.\langle 0, 0\rangle$ or $(\ldots, \lambda\overline{c_i}.\langle 0, 0\rangle, \ldots)$ according to the domain we are considering. In Fig. 13, we define a number of basic operations on the lam model that are used in the semantics of contracts.

The relevant property for the following theoretical development is the one below. We say that an operation is monotone if, whenever it is applied to arguments in the pre-order relation $\Subset$, it returns values in the same pre-order relation $\Subset$. The proof is straightforward and therefore omitted.

Proposition 4 *The operations of extension, parallel, sequence and plus are monotone with respect to* $\Subset$. *Additionally, if* $\mathcal{L} \Subset \mathcal{L}'$, *then* $\mathcal{L}[\overline{c'}/\overline{c}] \Subset \mathcal{L}'[\overline{c'}/\overline{c}]$.

5.2 The finite approximants of abstract method behaviors

As explained above, the lam model of a `core ABS` program is obtained by means of a fixpoint technique plus a saturation

$$
\begin{array}{lrcl}
\text{[extension]} & \mathcal{L}\&(c,c')^{[\mathtt{w}]} & \stackrel{def}{=} & \{L \cup \{(c,c')^{[\mathtt{w}]}\} \mid L \in \mathcal{L}\}. \\
\text{[parallel]} & \mathcal{L}\|\mathcal{L}' & \stackrel{def}{=} & \{L \cup L' \mid L \in \mathcal{L} \text{ and } L' \in \mathcal{L}'\} \\
\text{[extension (on pairs of lams)]} & \langle \mathcal{L}, \mathcal{L}' \rangle \& (c,c')^{[\mathtt{w}]} & \stackrel{def}{=} & \langle \mathcal{L}\&(c,c')^{[\mathtt{w}]}, \mathcal{L}' \rangle \\
\text{[parallel (on pairs of lams)]} & \langle \mathcal{L}_1, \mathcal{L}_1' \rangle \parallel \langle \mathcal{L}_2, \mathcal{L}_2' \rangle & \stackrel{def}{=} & \langle (\mathcal{L}_1 \cup \mathcal{L}_1') \| (\mathcal{L}_2 \cup \mathcal{L}_2'), 0 \rangle \\
\text{[sequence (on pairs of lams)]} & \langle \mathcal{L}_1, \mathcal{L}_1' \rangle \fatsemi \langle \mathcal{L}_2, \mathcal{L}_2' \rangle & \stackrel{def}{=} & \begin{cases} \langle \mathcal{L}_1, \mathcal{L}_1' \| \mathcal{L}_2' \rangle & \text{if } \mathcal{L}_2 = 0 \\ \langle \mathcal{L}_1 \cup (\mathcal{L}_2 \| \mathcal{L}_1'), \mathcal{L}_1' \| \mathcal{L}_2' \rangle & \text{otherwise} \end{cases} \\
\text{[plus (on pairs of lams)]} & \langle \mathcal{L}_1, \mathcal{L}_1' \rangle + \langle \mathcal{L}_2, \mathcal{L}_2' \rangle & \stackrel{def}{=} & \langle \mathcal{L}_1 \cup \mathcal{L}_2, \mathcal{L}_1' \cup \mathcal{L}_2' \rangle.
\end{array}
$$

Fig. 13 Lam operations

$$
\lceil \_ \rceil \stackrel{def}{=} \varepsilon \qquad \lceil X \rceil \stackrel{def}{=} \varepsilon \qquad \lceil [cog{:}c, x_1{:}\mathtt{r}_1, \cdots, x_n{:}\mathtt{r}_n] \rceil \stackrel{def}{=} c \lceil \mathtt{r}_1 \rceil \cdots \lceil \mathtt{r}_n \rceil \qquad \lceil c \rightsquigarrow \mathtt{r} \rceil \stackrel{def}{=} c \lceil \mathtt{r} \rceil \qquad \lceil \overline{\mathtt{r}}, \overline{\mathtt{s}} \rceil \stackrel{def}{=} \lceil \overline{\mathtt{r}} \rceil \lceil \overline{\mathtt{s}} \rceil
$$

Fig. 14 Extraction process

$$
(\_ \curvearrowright \_) \stackrel{def}{=} \varepsilon \qquad (\mathtt{r} \curvearrowright X) \stackrel{def}{=} \varepsilon \qquad ([cog{:}c', x_1{:}\mathtt{r}_1', \cdots, x_n{:}\mathtt{r}_n'] \curvearrowright [cog{:}c, x_1{:}\mathtt{r}_1, \cdots, x_n{:}\mathtt{r}_n]) \stackrel{def}{=} c' \, (\mathtt{r}_1' \curvearrowright \mathtt{r}_1) \cdots (\mathtt{r}_n' \curvearrowright \mathtt{r}_n)
$$
$$
(c' \rightsquigarrow \mathtt{r}' \curvearrowright c \rightsquigarrow \mathtt{r}) \stackrel{def}{=} c' \, (\mathtt{r}' \curvearrowright \mathtt{r})
$$

Fig. 15 Cog mapping process

applied to its contract class table. In particular, the lam model of the class table is a lam table that maps each method $\mathtt{C.m}$ of the program to $\lambda \overline{c}_{\mathtt{C.m}}.\langle \mathcal{L}_{\mathtt{C.m}}, \mathcal{L}'_{\mathtt{C.m}} \rangle$ where $\overline{c}_{\mathtt{C.m}} = \lceil \mathtt{r}, \overline{\mathtt{s}} \rceil$, with $\mathtt{r}(\overline{\mathtt{s}})$ being the header of the method contract of $\mathtt{C.m}$. The definition of $\lceil \mathtt{r}, \overline{\mathtt{s}} \rceil$ is given in Fig. 14 (we recall that names in headers occur linearly). The definition of the cog mapping process is given in Fig. 15. The following definition presents the algorithm used to compute the next approximant of a contract class table.

Definition 5 Let CCT be a contract class table of the form $(\ldots, \mathtt{C.m} \mapsto \mathtt{r}_{\mathtt{C.m}}(\overline{\mathtt{s}_{\mathtt{C.m}}}) \, \{\langle \mathbb{c}_{\mathtt{C.m}}, \mathbb{c}'_{\mathtt{C.m}} \rangle\} \, \mathtt{r}'_{\mathtt{C.m}}, \ldots)$

1. the *approximant 0* is defined as
$$(\ldots, \lambda(\lceil \mathtt{r}_{\mathtt{C.m}}, \overline{\mathtt{s}_{\mathtt{C.m}}} \rceil).\langle 0, 0 \rangle, \ldots) \; ;$$
2. let $\mathfrak{L} = \big(\ldots, \lambda \lceil \mathtt{r}_{\mathtt{C.m}}, \overline{\mathtt{s}_{\mathtt{C.m}}} \rceil.\langle \mathcal{L}_{\mathtt{C.m}}, \mathcal{L}'_{\mathtt{C.m}} \rangle, \ldots\big)$ be the n-th approximant; the $n+1$-th approximant is defined as $(\ldots, \lambda \lceil \mathtt{r}_{\mathtt{C.m}}, \overline{\mathtt{s}_{\mathtt{C.m}}} \rceil.\langle \mathcal{L}''_{\mathtt{C.m}}, \mathcal{L}'''_{\mathtt{C.m}} \rangle, \ldots)$ where $\langle \mathcal{L}''_{\mathtt{C.m}}, \mathcal{L}'''_{\mathtt{C.m}} \rangle = \mathbb{c}_{\mathtt{C.m}}(\mathfrak{L})_c \fatsemi \mathbb{c}'_{\mathtt{C.m}}(\mathfrak{L})_c$ with c being the cog of $\mathtt{r}_{\mathtt{C.m}}$ and the function $\mathbb{c}(\mathcal{L})_c$ being defined by structural induction in Fig. 16.

It is worth to notice that there are two rules for synchronous invocations in Fig. 16: (L- SInvk) dealing with synchronous invocations on the same cog name of the caller—the index c of the transformation, (L- RSInvk) dealing with synchronous invocations on different cog names.

Let
$$\Big(\ldots, \lambda \overline{c_{\mathtt{C,m}}}.\langle {\mathcal{L}_{\mathtt{C,m}}}^0, {\mathcal{L}'_{\mathtt{C,m}}}^0 \rangle, \ldots\Big) = \Big(\ldots, \lambda \overline{c_{\mathtt{C,m}}}.\langle 0, 0 \rangle\rangle, \ldots\Big),$$
and let
$$\Big(\ldots, \lambda \overline{c_{\mathtt{C,m}}}.\langle {\mathcal{L}_{\mathtt{C,m}}}^0, {\mathcal{L}'_{\mathtt{C,m}}}^0 \rangle, \ldots\Big),$$
$$\Big(\ldots, \lambda \overline{c_{\mathtt{C,m}}}.\langle {\mathcal{L}_{\mathtt{C,m}}}^1, {\mathcal{L}'_{\mathtt{C,m}}}^1 \rangle, \ldots\Big),$$
$$\Big(\ldots, \lambda \overline{c_{\mathtt{C,m}}}.\langle {\mathcal{L}_{\mathtt{C,m}}}^2, {\mathcal{L}'_{\mathtt{C,m}}}^2 \rangle, \ldots\Big), \cdots$$
be the sequence obtained by the algorithm of Definition 5 (this is the standard Knaster–Tarski technique). This sequence is non-decreasing (according to $\Subset$) because it is defined as a composition of monotone operators, see Proposition 5. Because of the creation of new cog names at each iteration, the fixpoint of the above sequence may not exist. We have already discussed the example of `Math.fact_nc`. In order to let our analysis terminate, after a given approximant, we run the Knaster–Tarski technique *using a different semantics for the operations* (L- SInvk), (L- RSInvk), (L- AInvk) *and* (L- GAInvk) (these are the rules where cog names may be created). In particular, when these operations are used at approximants larger than n, the renaming of free variables is disallowed. That is, the substitutions $[\overline{b'_{\mathtt{D,n}}}/\overline{b_{\mathtt{D,n}}}]$ in Fig. 16 are removed. It is straightforward to verify that these operations are still monotone. It is also straightforward to demonstrate by a simple cardinality argument the existence of fixpoints in the lam domain by running the Knaster–Tarski technique with this different semantics. This method is called a *saturation technique at n*.

For example, if we compute the third approximant of

$\mathtt{Math.fact\_nc} \mapsto$
$\quad [cog{:}c](\_)\{\langle 0 + \mathtt{Math!fact\_nc}\ [cog{:}c'](\_)$
$\quad \to \_.(c, c'), 0 \rangle\}\_$

1. let $\overline{b_{\mathtt{C,m}}} = (\mathit{cog\_names}(\mathcal{L}_{\mathtt{C,m}}) \cup \mathit{cog\_names}(\mathcal{L}'_{\mathtt{C,m}})) \setminus \overline{c_{\mathtt{C,m}}}$. These are the free cog names that are replaced by *fresh* cog names at every *transformation step*;
2. the transformation $\mathbb{c}_{\mathtt{C,m}}(\cdots \lambda\overline{c_{\mathtt{C,m}}}.\langle \mathcal{L}_{\mathtt{C,m}}, \mathcal{L}'_{\mathtt{C,m}}\rangle, \cdots)_c$ is defined inductively as follows:
   - $\langle 0, 0\rangle \& (c_1, c_2)^{[\mathtt{w}]}$ (L-GAZERO)
     if $\mathbb{c}_{\mathtt{C,m}} = (c_1, c_2)^{[\mathtt{w}]}$;
   - $(\lambda\overline{c_{\mathtt{D,n}}}.\langle \mathcal{L}_{\mathtt{D,n}}, \mathcal{L}'_{\mathtt{D,n}}\rangle \& (c,c)^{\mathtt{w}}[\overline{b'_{\mathtt{D,n}}}/\overline{b_{\mathtt{D,n}}}])(\mathbb{r}' \curvearrowright \mathbb{r}_{\mathtt{D,n}})(\overline{\mathbb{s}'} \curvearrowright \overline{\mathbb{s}_{\mathtt{D,n}}})$ (L-SINVK)
     if $\mathbb{c}_{\mathtt{C,m}} = \mathtt{D.n}\ \mathbb{r}'(\overline{\mathbb{s}'}) \to \mathbb{r}''$, $\mathbb{r}' = [\mathit{cog}{:}c, \overline{x}{:}\overline{\mathbb{r}'}]$, and $\mathrm{CCT}(\mathtt{D})(\mathtt{n}) = \ \mathbb{r}_{\mathtt{D,n}}(\overline{\mathbb{s}_{\mathtt{D,n}}})\ \{\langle \mathbb{c}_{\mathtt{D,n}}, \mathbb{c}'_{\mathtt{D,n}}\rangle\}\ \mathbb{r}'_{\mathtt{D,n}}$
     and $\overline{b'_{\mathtt{D,n}}}$ are fresh cog names;
   - $(\lambda\overline{c_{\mathtt{D,n}}}.\langle \mathcal{L}_{\mathtt{D,n}}, \mathcal{L}'_{\mathtt{D,n}}\rangle \& (c,c')[\overline{b'_{\mathtt{D,n}}}/\overline{b_{\mathtt{D,n}}}])(\mathbb{r}' \curvearrowright \mathbb{r}_{\mathtt{D,n}})(\overline{\mathbb{s}'} \curvearrowright \overline{\mathbb{s}_{\mathtt{D,n}}})$ (L-RSINVK)
     if $\mathbb{c}_{\mathtt{C,m}} = \mathtt{D.n}\ \mathbb{r}'(\overline{\mathbb{s}'}) \to \mathbb{r}''$, $\mathbb{r}' = [\mathit{cog}{:}c', \overline{x}{:}\overline{\mathbb{r}'}]$, and $c \neq c'$ and $\mathrm{CCT}(\mathtt{D})(\mathtt{n}) = \ \mathbb{r}_{\mathtt{D,n}}(\overline{\mathbb{s}_{\mathtt{D,n}}})\ \{\langle \mathbb{c}_{\mathtt{D,n}}, \mathbb{c}'_{\mathtt{D,n}}\rangle\}\ \mathbb{r}'_{\mathtt{D,n}}$
     and $\overline{b'_{\mathtt{D,n}}}$ are fresh cog names;
   - $\langle 0, \mathcal{L}''_{\mathtt{D,n}} \cup \mathcal{L}'''_{\mathtt{D,n}}\rangle$ (L-AINVK)
     if $\mathbb{c}_{\mathtt{C,m}} = \mathtt{D!n}\ \mathbb{r}'(\overline{\mathbb{s}'}) \to \mathbb{r}''$ and $\mathrm{CCT}(\mathtt{D})(\mathtt{n}) = \ \mathbb{r}_{\mathtt{D,n}}(\overline{\mathbb{s}_{\mathtt{D,n}}})\ \{\langle \mathbb{c}_{\mathtt{D,n}}, \mathbb{c}'_{\mathtt{D,n}}\rangle\}\ \mathbb{r}'_{\mathtt{D,n}}$
     and $\langle \mathcal{L}''_{\mathtt{D,n}}, \mathcal{L}'''_{\mathtt{D,n}}\rangle = (\lambda\overline{c_{\mathtt{D,n}}}.\langle \mathcal{L}_{\mathtt{D,n}}, \mathcal{L}'_{\mathtt{D,n}}\rangle[\overline{b'_{\mathtt{D,n}}}/\overline{b_{\mathtt{D,n}}}])(\mathbb{r}' \curvearrowright \mathbb{r}_{\mathtt{D,n}})(\overline{\mathbb{s}'} \curvearrowright \overline{\mathbb{s}_{\mathtt{D,n}}})$ and $\overline{b'_{\mathtt{D,n}}}$ are fresh cog names;
   - $(\lambda\overline{c_{\mathtt{D,n}}}.\langle \mathcal{L}_{\mathtt{D,n}}, \mathcal{L}'_{\mathtt{D,n}}\rangle \& (c_1, c_2)^{[\mathtt{w}]}[\overline{b'_{\mathtt{D,n}}}/\overline{b_{\mathtt{D,n}}}])(\mathbb{r}' \curvearrowright \mathbb{r}_{\mathtt{D,n}})(\overline{\mathbb{s}'} \curvearrowright \overline{\mathbb{s}_{\mathtt{D,n}}})$ (L-GAINVK)
     if $\mathbb{c}_{\mathtt{C,m}} = \mathtt{D!n}\ \mathbb{r}'(\overline{\mathbb{s}'}) \to \mathbb{r}''.(c_1, c_2)^{[\mathtt{w}]}$ and $\mathrm{CCT}(\mathtt{D})(\mathtt{n}) = \ \mathbb{r}_{\mathtt{D,n}}(\overline{\mathbb{s}_{\mathtt{D,n}}})\ \{\langle \mathbb{c}_{\mathtt{D,n}}, \mathbb{c}'_{\mathtt{D,n}}\rangle\}\ \mathbb{r}'_{\mathtt{D,n}}$ and $\overline{b'_{\mathtt{D,n}}}$ are fresh cog names;
   - $\mathbb{c}'_{\mathtt{C,m}}(\cdots, \lambda\overline{c_{\mathtt{C,m}}}.\langle \mathcal{L}_{\mathtt{C,m}}, \mathcal{L}'_{\mathtt{C,m}}\rangle, \cdots)_c \fatsemi \mathbb{c}''_{\mathtt{C,m}}(\cdots, \lambda\overline{c_{\mathtt{C,m}}}.\langle \mathcal{L}_{\mathtt{C,m}}, \mathcal{L}'_{\mathtt{C,m}}\rangle, \cdots)_c$ (L-SEQ)
     if $\mathbb{c}_{\mathtt{C,m}} = \mathbb{c}'_{\mathtt{C,m}} \fatsemi \mathbb{c}''_{\mathtt{C,m}}$;
   - $\mathbb{c}'_{\mathtt{C,m}}(\cdots, \lambda\overline{c_{\mathtt{C,m}}}.\langle \mathcal{L}_{\mathtt{C,m}}, \mathcal{L}'_{\mathtt{C,m}}\rangle, \cdots)_c + \mathbb{c}''_{\mathtt{C,m}}(\cdots, \lambda\overline{c_{\mathtt{C,m}}}.\langle \mathcal{L}_{\mathtt{C,m}}, \mathcal{L}'_{\mathtt{C,m}}\rangle, \cdots)_c$ (L-PLUS)
     if $\mathbb{c}_{\mathtt{C,m}} = \mathbb{c}'_{\mathtt{C,m}} + \mathbb{c}''_{\mathtt{C,m}}$;
   - $\mathbb{c}'_{\mathtt{C,m}}(\cdots, \lambda\overline{c_{\mathtt{C,m}}}.\langle \mathcal{L}_{\mathtt{C,m}}, \mathcal{L}'_{\mathtt{C,m}}\rangle, \cdots)_c \parallel \mathbb{c}''_{\mathtt{C,m}}(\cdots, \lambda\overline{c_{\mathtt{C,m}}}.\langle \mathcal{L}_{\mathtt{C,m}}, \mathcal{L}'_{\mathtt{C,m}}\rangle, \cdots)_c$ (L-PAR)
     if $\mathbb{c}_{\mathtt{C,m}} = \mathbb{c}'_{\mathtt{C,m}} \parallel \mathbb{c}''_{\mathtt{C,m}}$.

Fig. 16 Lam transformation of CCT

we get the sequence

$$\lambda c.\langle 0, 0\rangle$$
$$\lambda c.\langle\, [\,(c, c_0)\,]\,, 0\rangle$$
$$\lambda c.\langle\, [\,(c, c_0), (c_0, c_1)\,]\,, 0\rangle$$
$$\lambda c.\langle\, [\,(c, c_0), (c_0, c_1), (c_1, c_2)\,]\,, 0\rangle$$

and, if we saturate a this point, we obtain

$$\lambda c.\langle\, [\,(c, c_0)(c_0, c_0), (c_0, c_1), (c_1, c_2)\,]\,, 0\rangle$$
$$\lambda c.\langle\, [\,(c, c_0)(c_0, c_0), (c_0, c_1), (c_1, c_2)\,]\,, 0\rangle \qquad \text{fixpoint}$$

Definition 6 Let $(\mathrm{CT}, \{\overline{T\ x\ ;}\ s\}, \mathrm{CCT})$ be a `core ABS` program and let $\left(\ldots \lambda\overline{c_{\mathtt{C,m}}}.\langle {\mathcal{L}_{\mathtt{C,m}}}^{n+h}, {\mathcal{L}'_{\mathtt{C,m}}}^{n+h}\rangle, \ldots\right)$ be the fixpoint (unique upto renaming of cog names) obtained by the saturation technique at n. The *abstract class table at* n, written $\mathrm{ACT}_{[n]}$, is a map that takes `C.m` and returns $\lambda\overline{c_{\mathtt{C,m}}}.\langle {\mathcal{L}_{\mathtt{C,m}}}^{n+h}, {\mathcal{L}'_{\mathtt{C,m}}}^{n+h}\rangle$.

Let $(\mathrm{CT}, \{\overline{T\ x\ ;}\ s\}, \mathrm{CCT})$ be a `core ABS` program and

$$\Gamma \vdash_{\text{start}} \{\overline{T\ x\ ;}\ s\} : \langle \mathbb{c}, \mathbb{c}'\rangle \ \triangleright\ \mathcal{U} \mid \Gamma\ .$$

The *abstract semantics saturated at* n of $(\mathrm{CT}, \{\overline{T\ x\ ;}\ s\}, \mathrm{CCT})$ is $(\mathbb{c}(\mathrm{ACT}_{[n]})_{\text{start}}) \fatsemi (\mathbb{c}'(\mathrm{ACT}_{[n]})_{\text{start}})$.

As an example, in Fig. 17, we compute the abstract semantics saturated at 2 of the class `Math` in Fig. 5.

5.3 Deadlock analysis of lams

Definition 7 Let L be a relation on cog names (pairs in L are either of the form (c_1, c_2) or of the form $(c_1, c_2)^{\mathtt{w}}$. L *contains a circularity* if $L^{\mathtt{get}}$ has a pair (c, c) (see Definition 2). Similarly, $\langle \mathcal{L}, \mathcal{L}'\rangle$ (or $\lambda\overline{c}.\langle \mathcal{L}, \mathcal{L}'\rangle$) has a circularity if there is $L \in \mathcal{L} \cup \mathcal{L}'$ that contains a circularity.

A `core ABS` program with an abstract class table saturated at n is *deadlock-free* if its abstract semantics $\langle \mathcal{L}, \mathcal{L}'\rangle$ does not contain a circularity.

The fixpoints for `Math.fact_g` and `Math.fact_ag` are found at the third iteration. According to the above definition of deadlock freedom, `Math.fact_g` yields a deadlock, while `Math.fact_ag` is deadlock-free because $\{(c, c)^{\mathtt{w}}\}^{\mathtt{get}}$ does not contain any circularity. As discussed before, there exists no fixpoint for `Math.fact_nc`. If we decide to stop at the approximant 2 and saturate, we get

$$\lambda c.\langle\, [\,(c, c'), (c', c'), (c', c'')\,]\,, 0\rangle,$$

which contains a circularity that is a false positive.

Note that saturation might even start at the approximant 0 (where every method is $\lambda c.\langle 0, 0\rangle$). 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

$$\lambda c.\langle\, [\,(c, c'), (c', c')\,]\,, 0\rangle,$$

method	approx. 0	approx. 1	approx.2	saturation
`Math.fact_g`	$\lambda c.\langle 0, 0\rangle$	$\lambda c.\langle [(c,c)], 0\rangle$	$\lambda c.\langle [(c,c)], 0\rangle$	
`Math.fact_ag`	$\lambda c.\langle 0, 0\rangle$	$\lambda c.\langle [(c,c)^{\mathtt{w}}], 0\rangle$	$\lambda c.\langle [(c,c)^{\mathtt{w}}], 0\rangle$	
`Math.fact_nc`	$\lambda c.\langle 0, 0\rangle$	$\lambda c.\langle [(c,c')], 0\rangle$	$\lambda c.\langle [(c,c'),(c',c'')], 0\rangle$	$\lambda c.\langle [(c,c'),(c',c'),(c',c'')], 0\rangle$

Fig. 17 Abstract class table computation for class `Math`

method	approx. 0	approx. 1	approx.2
`CpxSched.m1`	$\lambda c,c',c''.\langle 0, 0\rangle$	$\lambda c,c',c''.\langle 0, [(c',c''),(c'',c')]\rangle$	$\lambda c,c',c''.\langle 0, [(c',c''),(c'',c')]\rangle$
`CpxSched.m2`	$\lambda c,c'.\langle 0, 0\rangle$	$\lambda c,c'.\langle [(c,c')], 0\rangle$	$\lambda c,c'.\langle [(c,c')], 0\rangle$
`CpxSched.m3`	$\lambda c.\langle 0, 0\rangle$	$\lambda c.\langle 0, 0\rangle$	

Fig. 18 Abstract class table computation for class `CpxSched`

contract pairs

$$\mathbb{cp} \ ::= \ \mathbb{c} \ \mid \ \langle \mathbb{cp}, \mathbb{cp}\rangle_c \ \mid \ \mathbb{cp}\&(c,c')^{[\mathtt{w}]} \ \mid \ \mathbb{cp} + \mathbb{cp} \ \mid \ \mathbb{cp} \mathbin{;} \mathbb{cp} \ \mid \ \mathbb{cp} \parallel \mathbb{cp}$$

contract pairs contexts ($\sharp \in \{+, ;, \parallel\}$)

$$\mathfrak{D}[\,] \ ::= \ [\,] \ \mid \ \mathfrak{D}[\,]\&(c',c'')^{[\mathtt{w}]} \ \mid \ \mathfrak{D}[\,]\sharp\mathbb{cp} \ \mid \ \mathbb{cp}\sharp\mathfrak{D}[\,]$$

$$\mathfrak{C}[\,]_c \ ::= \ \langle \mathfrak{D}[\,], \mathbb{cp}\rangle_c \ \mid \ \langle \mathbb{cp}, \mathfrak{D}[\,]\rangle_c \ \mid \ \langle \mathfrak{C}[\,]_c, \mathbb{cp}\rangle_{c'} \ \mid \ \langle \mathbb{cp}, \mathfrak{C}[\,]_c\rangle_{c'} \ \mid \ \mathfrak{C}[\,]_c\&(c',c'')^{[\mathtt{w}]} \ \mid \ \mathfrak{C}[\,]_c\sharp\mathbb{cp} \ \mid \ \mathbb{cp}\sharp\mathfrak{C}[\,]_c$$

reduction relation

(Red-SInvk)

$$\frac{\begin{array}{c}\mathtt{C.m} = \mathbb{s}(\overline{\mathbb{s}})\{\langle \mathbb{c}, \mathbb{c}'\rangle\}\mathbb{s}' \qquad \mathbb{r} = [cog{:}c, \overline{x}{:}\overline{\mathbb{r}''}] \\ cog\_names(\langle \mathbb{c}, \mathbb{c}'\rangle) \setminus cog\_names(\mathbb{s}, \overline{\mathbb{s}}) = \widetilde{z} \\ \widetilde{w} \text{ are fresh} \qquad \langle \mathbb{c}, \mathbb{c}'\rangle[{}^{\widetilde{w}}/_{\widetilde{z}}][{}^{\mathbb{r}, \overline{\mathbb{r}}}/_{\mathbb{s}, \overline{\mathbb{s}}}] = \langle \mathbb{c}'', \mathbb{c}'''\rangle\end{array}}{\mathfrak{C}[\mathtt{C.m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}']_c \longrightarrow \mathfrak{C}[\langle \mathbb{c}'', \mathbb{c}'''\rangle_c]_c}$$

(Red-RSInvk)

$$\frac{\begin{array}{c}\mathtt{C.m} = \mathbb{s}(\overline{\mathbb{s}})\{\langle \mathbb{c}, \mathbb{c}'\rangle\}\mathbb{s}' \qquad \mathbb{r} = [cog{:}c', \overline{x}{:}\overline{\mathbb{r}''}] \qquad c \neq c' \\ cog\_names(\langle \mathbb{c}, \mathbb{c}'\rangle) \setminus cog\_names(\mathbb{s}, \overline{\mathbb{s}}) = \widetilde{z} \\ \widetilde{w} \text{ are fresh} \qquad \langle \mathbb{c}, \mathbb{c}'\rangle[{}^{\widetilde{w}}/_{\widetilde{z}}][{}^{\mathbb{r}, \overline{\mathbb{r}}}/_{\mathbb{s}, \overline{\mathbb{s}}}] = \langle \mathbb{c}'', \mathbb{c}'''\rangle\end{array}}{\mathfrak{C}[\mathtt{C.m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}']_c \longrightarrow \mathfrak{C}[\langle \mathbb{c}'', \mathbb{c}'''\rangle_{c'}\&(c,c')]_c}$$

(Red-AInvk)

$$\frac{\begin{array}{c}\mathtt{C.m} = \mathbb{s}(\overline{\mathbb{s}})\{\langle \mathbb{c}, \mathbb{c}'\rangle\}\mathbb{s}' \qquad \mathbb{r} = [cog{:}c', \overline{x}{:}\overline{\mathbb{r}''}] \\ cog\_names(\langle \mathbb{c}, \mathbb{c}'\rangle) \setminus cog\_names(\mathbb{s}, \overline{\mathbb{s}}) = \widetilde{z} \\ \widetilde{w} \text{ are fresh} \qquad \langle \mathbb{c}, \mathbb{c}'\rangle[{}^{\widetilde{w}}/_{\widetilde{z}}][{}^{\mathbb{r}, \overline{\mathbb{r}}}/_{\mathbb{s}, \overline{\mathbb{s}}}] = \langle \mathbb{c}'', \mathbb{c}'''\rangle\end{array}}{\mathfrak{C}[\mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}']_c \longrightarrow \mathfrak{C}[\langle \mathbb{c}'', \mathbb{c}'''\rangle_{c'}]_c}$$

(Red-GAInvk)

$$\frac{\begin{array}{c}\mathtt{C.m} = \mathbb{s}(\overline{\mathbb{s}})\{\langle \mathbb{c}, \mathbb{c}'\rangle\}\mathbb{s}' \qquad \mathbb{r} = [cog{:}c', \overline{x}{:}\overline{\mathbb{r}''}] \\ cog\_names(\langle \mathbb{c}, \mathbb{c}'\rangle) \setminus cog\_names(\mathbb{s}, \overline{\mathbb{s}}) = \widetilde{z} \\ \widetilde{w} \text{ are fresh} \qquad \langle \mathbb{c}, \mathbb{c}'\rangle[{}^{\widetilde{w}}/_{\widetilde{z}}][{}^{\mathbb{r}, \overline{\mathbb{r}}}/_{\mathbb{s}, \overline{\mathbb{s}}}] = \langle \mathbb{c}'', \mathbb{c}'''\rangle\end{array}}{\mathfrak{C}[\mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}'.(c'',c''')^{[\mathtt{w}]}]_c \longrightarrow \mathfrak{C}[\langle \mathbb{c}'', \mathbb{c}'''\rangle_{c'}\&(c'',c''')^{[\mathtt{w}]}]_c}$$

Fig. 19 Contract reduction rules

which contains a circularity.

In general, in techniques like the one we have presented, it is possible to augment the precision of the analysis by delaying the saturation. However, assuming that pairwise different method contracts have disjoint free cog names (which is a reasonable assumption), we have not found any sample `core ABS` code where saturating at 1 gives a better precision than saturating at 0. While this issue is left open, the current version of our tool `DF4ABS` allows one to specify the saturation point; the default saturation point is 0.

The computation of the abstract class table for class `CpxSched` does not need any saturation, all methods are non-recursive and encounter their fixpoint by iteration 2 (see Fig. 18). The abstract class table shows a circularity for method `m1`, manifesting the presence of a deadlock.

The correctness of the fixpoint analysis of contracts discussed in this section is demonstrated in "Appendix 2".

We remark that this technique is as modular as the inference system: Once the contracts of a module have been computed, one may run the fixpoint analysis and attach the corresponding abstract values to the code. Analyzing a program reduces to computing the lam of the main function.

6 The model-checking analysis of contracts

The second analysis technique for the contracts of Sect. 4 consists of computing contract models *by expanding* their invocations. We therefore begin this section by introducing a semantics of contracts that is alternative to the one of Sect. 5.

6.1 Operational semantics of contracts

The operational semantics of a contract is defined as a reduction relation between terms that are *contract pairs* $\mathbb{cp}$, whose syntax is defined in Fig. 19. These contract pairs highlight (in the operational semantics) the fact that every contract actually represents two collections of relations on cog names: those corresponding to the *present states* and those corresponding to *future states*. We have discussed this dichotomy in Sect. 4.

In Fig. 19, we have also defined the *contract pair contexts*, noted $\mathfrak{C}[\ ]_c$, which are indexed contract pairs with a hole. The index c indicates that the hole is immediately enclosed by $\langle\cdot,\cdot\rangle_c$.

The reduction relation that defines the evaluation of contract pairs $\langle \mathbb{cp}_1, \mathbb{cp}_1' \rangle_c \longrightarrow \langle \mathbb{cp}_2, \mathbb{cp}_2' \rangle_c$ is defined in Fig. 19. There are four reduction rules: (RED- SINVK) for synchronous invocation on the same cog name of the caller (which is stored in the index of the enclosing pair), (RED- RSINVK) for synchronous invocations on different cog name, (RED- AINVK) for asynchronous invocations, and (RED- GAINVK) for asynchronous invocations with synchronizations. We observe that every evaluation step amounts to expanding method invocations by replacing free cog names in method contracts with fresh names and *without modifying the syntax tree of contract pairs*.

To illustrate the operational semantics of contracts, we discuss three examples:

1. Let
   $\mathtt{F.f} = [cog : c](x : [cog : c'], y : [cog : c'']) \{$
   $\langle (\mathtt{F.g}\, [cog : c'](x : [cog : c'']) \rightarrow \_).(c, c') + 0.(c', c''), 0\rangle$
   $\}\_$

   and
   $\mathtt{F.g} = [cog : c](x : [cog : c'])\{\langle 0.(c, c') + 0.(c', c), 0\rangle\}\_$

   Then

   $\langle \mathtt{F!f}[cog : c](x : [cog : c'], y : [cog : c'']) \rightarrow \_, 0\rangle_{\text{start}}$
   $\longrightarrow \langle\langle 0, (\mathtt{F.g}[cog : c'](x : [cog : c'']) \rightarrow \_).(c, c') + 0.(c', c'')\rangle_c, 0\rangle_{\text{start}}$
   $\longrightarrow \langle\langle 0, \langle 0.(c', c'') + 0.(c'', c'), 0\rangle_{c'} \& (c, c') + 0.(c', c'')\rangle_c, 0\rangle_{\text{start}}$

   The contract pair in the final state *does not contain method invocations*. This is because the above main function is not recursive. Additionally, the evaluation of $\mathtt{F.f}\, [cog : c](x : [cog : c'], y : [cog : c''])$ *has not created names*. This is because names in the bodies of `F.f` and `F.g` are bound.
2. Let

   $\mathtt{F.h} = [cog : c](\_)\{\langle 0, (\mathtt{F.h}[cog : c'](\_) \rightarrow \_) \| 0.(c, c')\rangle\}\_$

   Then
   $\langle \mathtt{F!h}\, [cog : c](\_) \rightarrow \_, 0\rangle_{\text{start}}$
   $\longrightarrow \langle\langle 0, (\mathtt{F.h}[cog : c'](\_) \rightarrow \_) \| 0.(c, c')\rangle_c, 0\rangle_{\text{start}}$
   $\longrightarrow \langle\langle 0, \langle 0, (\mathtt{F.h}[cog : c''](\_) \rightarrow \_) \| 0.(c', c'')\rangle_{c'} \| 0.(c, c')\rangle_c, 0\rangle_{\text{start}}$
   $\longrightarrow \cdots$

   where, in this case, the contract pairs grow in the number of dependencies as the evaluation progresses. This growth *is due to the presence of a free name* in the definition of `F.h` that, as said, corresponds to generating a fresh name at every recursive invocation.
3. Let

   $\mathtt{F.l} = [cog : c]()\{\langle 0.(c, c') \fatsemi (0.(c, c') \| \mathtt{F!l}\, [cog : c]()$
   $\rightarrow \_), 0\rangle\}\_$

   Then

   $\langle \mathtt{F!l}\, [cog : c]() \rightarrow \_, 0\rangle_{\text{start}}$
   $\longrightarrow \langle\langle 0, 0.(c, c') \fatsemi (0.(c, c') \| \mathtt{F!l}\, [cog : c]() \rightarrow \_)\rangle_c, 0\rangle_{\text{start}}$
   $\longrightarrow \langle\langle 0, 0.(c, c') \fatsemi (0.(c, c') \|$
   $\langle 0, 0.(c', c'') \fatsemi (0.(c', c'') \| \mathtt{F!l}\, [cog : c''](\,) \rightarrow \_)\rangle_{c'})\rangle_c, 0\rangle_{\text{start}}$
   $\longrightarrow \cdots$

   In this case, the contract pairs grow in the number of "$\fatsemi$"-terms, which become larger and larger as the evaluation progresses.

It is clear that, in the presence of recursion and of free cog names in method contracts, a technique that analyses contracts by expanding method invocations is fated to fail because the system contains infinite states. However, it is possible to stop the expansions at suitable points without losing any relevant information about dependencies. In this section, we highlight the technique we have developed in [18] that has been prototyped for `core ABS` in `DF4ABS`.

6.2 Linear recursive contract class tables

Since contract pairs models may contain infinite states, instead of resorting to a saturation technique, which introduces inaccuracies, we exploit a generalization of permutation theory that let us decide when stopping the evaluation with the guarantee that if no circular dependency has been found upto that moment, then it will not appear afterward. That stage corresponds to the *order* of an associated permutation. It turns out that this technique is suited for so-called *linear recursive* contract class tables.

Definition 8 A contract class table is *linear recursive* if (mutual) recursive invocations in bodies of methods have *at most one recursive invocation*.

It is worth to observe that a `core ABS` program may be linear recursive, while the corresponding contract class table is not. For example, consider the following method `foo` of

$$\begin{aligned}
&\texttt{Foo.foo} = [cog : c](\_, x : [cog : c'])\{\ \langle 0\ +\ \Big(\big(\mathbb{c}.(c, c')^{\mathtt{w}} \parallel \mathbb{c}'\big) \fatsemi \mathbb{c}'.(c, c)^{\mathtt{w}} \Big), 0 \rangle\ \} \to \_ \\
&\text{where}\ \ \mathbb{c} = \texttt{Print!print}\ [cog : c'](\_) \to \_ \\
&\qquad\quad \mathbb{c}' = \texttt{Foo!foo}\ [cog : c](\_, x : [cog : c']) \to \_
\end{aligned}$$

Fig. 20 Method contract of Foo.foo

class Foo that prints integers by invoking a printer service and awaits for the termination of the printer task and for its own termination:

```
Void foo(Int n, Print x){
    Fut<Void> u, v ;
    if (n == 0) return() ;
    else { u = this!foo(n-1, x) ;
           v = x!print(n) ;
           await v? ;
           await u? ;
           return() ;
     }
}
```

While foo has only one recursive invocation, its contract written in Fig. 20 is not. That is, the contract of Foo.foo displays *two* recursive invocations because, in correspondence of the await v? instruction, we need to collect all the effects produced by the previous unsynchronized asynchronous invocations [see rule (T- AWAIT)].[3]

6.3 Mutations and flashbacks

The idea of our technique is to consider the patterns of cog names in the formal parameters and the (at most unique) recursive invocation of method contracts and to study the changes. For example, the above method contracts of F.h and F.l transform the pattern of cog names in the formal parameters, written (c) into the pattern of recursive invocation (c'). We write this transformation as

$(c) \ \leadsto\ (c')$.

In general, the transformations we consider are called *mutations*.

Definition 9 A *mutation* is a transformation of tuples of (cog) names, written

$(x_1, \ldots, x_n) \ \leadsto\ (x'_1, \ldots, x'_n)$

where $x_1, \ldots, x_n$ are pairwise different and x'_i *may not occur in* $\{x_1, \ldots, x_n\}$.

[3] It is possible to define sufficient conditions on core ABS programs that entail linear recursive contract class tables. For example, two such conditions are that, in (mutual) recursive methods, recursive invocations are either (1) synchronous or (2) asynchronous followed by a get or await synchronization on the future value, without any other get or await synchronization or synchronous invocation in between.

Applying a mutation $(x_1, \ldots, x_n) \leadsto (x'_1, \ldots, x'_n)$ to a tuple of cog names (that may contain duplications) $(c_1, \ldots, c_n)$ gives a tuple $(c'_1, \ldots, c'_n)$ where

- $c'_i = c_j$ if $x'_i = x_j$;
- c'_i is a fresh name if $x'_i \notin \{x_1, \ldots, x_n\}$;
- $c'_i = c'_j$ if they are both fresh and $x'_i = x'_j$.

We write $(c_1, \ldots, c_n) \rightarrow_{\mathtt{mut}} (c'_1, \ldots, c'_n)$ when $(c'_1, \ldots, c'_n)$ is obtained by applying a mutation (which is kept implicit) to $(c_1, \ldots, c_n)$.

For example, given the mutation

$$(x, y, z, u) \leadsto (y, x, z', z') \tag{1}$$

we obtain the following sequence of tuples:

$$\begin{aligned}
(c, c', c'', c''') &\rightarrow_{\mathtt{mut}} (c', c, c_1, c_1) \\
&\rightarrow_{\mathtt{mut}} (c, c', c_2, c_2) \\
&\rightarrow_{\mathtt{mut}} (c', c, c_3, c_3) \\
&\rightarrow_{\mathtt{mut}} \quad \cdots
\end{aligned} \tag{2}$$

When a mutation $(x_1, \ldots, x_n) \leadsto (x'_1, \ldots, x'_n)$ is such that $\{x_1, \ldots, x_n\} = \{x'_1, \ldots, x'_n\}$, then the mutation is a *permutation* [6]. In this case, the permutation theory guarantees that, by repeatedly applying the same permutation to a tuple of names, at some point, one obtains the initial tuple. This point, which is known as the *order of the permutation*, allows one to define the following algorithm for linear recursive method contracts whose mutation is a permutation:

1. compute the order of the permutation associated with the recursive method contract and
2. correspondingly unfold the term to evaluate.

It is clear that, when method contract bodies have no free cog names, further unfoldings of the recursive method contract cannot add new dependencies. Therefore, the evaluation, as far as dependencies are concerned, may stop.

When a mutation is not a permutation, as in the example above, it is not possible to get again an old tuple by applying the mutation because of the presence of fresh names. However, it is possible to demonstrate that a tuple is equal to an old one *upto* a suitable map, called flashback.

Fig. 21 Reduction for contract of method `Math.fact_nc`

$$\begin{array}{ll}\langle \texttt{Math!fact\_nc}\,[cog : c](\_) \to \_, 0\rangle_{\texttt{start}} & \\ \longrightarrow & \langle\langle 0, 0 + \texttt{Math!fact\_nc}[cog : c'](\_) \to \_.(c, c')\rangle_{c}, 0\rangle_{\texttt{start}} \\ \longrightarrow & \langle\langle 0, 0 + \langle 0, 0 + \texttt{Math!fact\_nc}[cog : c''](\_) \to \_.(c', c'')\rangle_{c'}.(c, c')\rangle_{c}, 0\rangle_{\texttt{start}}\end{array}$$

Fig. 22 Flattening and evaluation of resulting contract of method `Math.fact_nc`

$$\begin{array}{l}(\llbracket\langle\langle 0, 0 + \langle 0, 0 + \texttt{Math!fact\_nc}[cog : c''](\_) \to \_.(c', c'')\rangle_{c'}.(c, c')\rangle_{c}, 0\rangle_{\texttt{start}}\rrbracket)^{\flat} \\ = (\langle\langle\langle 0, 0 + \langle 0, 0 + \langle\, [(c', c'')]\, , 0\rangle\rangle.(c, c')\rangle, 0\rangle)^{\flat} \\ = \langle 0 + 0 + \,[(c', c'')]\, \&(c, c'), 0\rangle \\ = \langle\, [(c', c''), (c, c')]\, , 0\rangle\end{array}$$

Definition 10 A tuple of cog names $(c_1, \ldots, c_n)$ is equivalent to $(c'_1, \ldots, c'_n)$, written $(c_1, \ldots, c_n) \approx (c'_1, \ldots, c'_n)$, if there is an injection $\imath$ called *flashback* such that:

1. $(c_1, \ldots, c_n) = (\imath(c'_1), \ldots, \imath(c'_n))$
2. $\imath$ is the identity on "old names", that is, if $c'_i \in \{c_1, \ldots, c_n\}$ then $\imath(c'_i) = c'_i$.

For example, in the sequence of transitions (2), there is a flashback from the last tuple to the second one and there is [and there will be, by applying the mutation (1)] no tuple that is equivalent to the initial tuple.

It is possible to generalize the result about permutation orders:

Theorem 1 (Giachino and Laneve [18]) *Let* $(x_1, \ldots, x_n) \rightsquigarrow (x'_1, \ldots, x'_n)$ *be a mutation and let*

$$(c_1, \ldots, c_n) \to_{\texttt{mut}} (c_{n+1}, \ldots, c_{2n}) \to_{\texttt{mut}} (c_{2n+1}, \ldots, c_{3n}) \to_{\texttt{mut}} \cdots$$

be a sequence of applications of the mutation. Then, there are $0 \leq h < k$ *such that*

$$(c_{hn+1}, \ldots, c_{(h+1)n}) \approx (c_{kn+1}, \ldots, c_{(k+1)n})$$

The value k is called order of the mutation.

For example, the order of the mutation (1) is 3.

6.4 Evaluation of the main contract pair

The generalization of permutation theory in Theorem 1 allows us to define the notion of *order of the contract of the main function* in a linear recursive contract class table. This order is the length of the evaluation of the contract obtained

1. by unfolding every recursive function as many times as *twice its ordering*,[4]
2. by iteratively applying 1 to every invocation of recursive function that has been produced during the unfolding.

[4] The interested reader may find in [18] the technical reason for unfolding recursive methods as many times as twice the length of the order of the corresponding mutation.

In order to state the theorem of the correctness of our analysis technique, we need to define the lam of a contract pair. The following functions will do the job.

Let $\llbracket\cdot\rrbracket$ be a map taking a contract pair and returning a pair of lams that is defined by

$$\begin{array}{l}\llbracket \texttt{C.m}\ \mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{r}'\rrbracket = \langle 0, 0\rangle \\ \llbracket \texttt{C!m}\ \mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{r}'\rrbracket = \langle 0, 0\rangle \\ \llbracket \texttt{C!m}\ \mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{r}'.(c, c')^{[\mathtt{w}]}\rrbracket = \langle\, [\,(c, c')^{[\mathtt{w}]}\,]\, , 0\rangle \\ \llbracket\langle \mathbb{cp}, \mathbb{cp}'\rangle_{c}\rrbracket = \langle\llbracket\mathbb{cp}\rrbracket, \llbracket\mathbb{cp}'\rrbracket\rangle\end{array}$$

and it is homomorphic with respect to the operations $+$, $\fatsemi$, $\parallel$ (whose definition on pairs of lams is in Fig. 13).

Let $\mathbb{t}$ be terms of the following syntax

$$\mathbb{t} ::= \quad \mathcal{L} \quad | \quad \langle \mathbb{t}, \mathbb{t}\rangle$$

and let $(\mathcal{L})^{\flat} = \mathcal{L}$ and $(\langle \mathbb{t}, \mathbb{t}'\rangle)^{\flat} = (\mathbb{t})^{\flat},\ (\mathbb{t}')^{\flat}$.

Theorem 2 (Giachino and Laneve [18]) *Let* $\langle \mathbb{cp}_1, \mathbb{cp}'_1\rangle$ *be a main function contract and let*

$$\langle \mathbb{cp}_1, \mathbb{cp}'_1\rangle_{\mathsf{start}} \longrightarrow \langle \mathbb{cp}_2, \mathbb{cp}'_2\rangle_{\mathsf{start}} \longrightarrow \langle \mathbb{cp}_3, \mathbb{cp}'_3\rangle_{\mathsf{start}} \longrightarrow \cdots$$

be its evaluation. Then, there is a k, which is the order of $\langle \mathbb{cp}_1, \mathbb{cp}'_1\rangle_{\mathsf{start}}$ *such that if a circularity occurs in* $(\llbracket\langle \mathbb{cp}_{k+h}, \mathbb{cp}'_{k+h}\rangle_{\mathsf{start}}\rrbracket)^{\flat}$*, for every h, then it also occurs in* $(\llbracket\langle \mathbb{cp}_{k}, \mathbb{cp}'_{k}\rangle_{\mathsf{start}}\rrbracket)^{\flat}$.

Example 7 The reduction of the contract of method `Math.fact_nc` is as in Fig. 21. The theory of mutations provide us with an order for this evaluation. In particular, the mutation associated with `Math.fact_nc` is $c \rightsquigarrow c'$, with order 1, such that after one step, we can encounter a flashback to a previous state of the mutation. Therefore, we need to reduce our contract for a number of steps corresponding to twice the ordering of `Math.fact_nc`: After two steps, we find the flashback associating the last generated pair (c', c'') with the one produced in the previous step (c, c'), by mapping c' to c and c'' to c'.

The flattening and the evaluation of the resulting contract are shown in Fig. 22 and produce the pair of lams $\langle\, [(c', c''), (c, c')]\, , 0\rangle$ which does not present any deadlock. Thus, differently from the fixpoint analysis for the same example, with this operational analysis, we get a precise answer instead of a false positive (see Fig. 17; Sect. 5.3).

The correctness of the technique based on mutations is demonstrated in "Appendix 3".

7 The DF4ABS tool and its application to the case study

core ABS (actually full ABS [22]) comes with a suite [39] 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 deadlock analysis framework (at the time of writing, the suite has only the fixpoint analyzer, the full framework is available at http://df4abs.nws.cs.unibo.it). The DF4ABS tool is built upon the abstract syntax tree (AST) of the core ABS type checker, which allows us to exploit the type information stored in every node of the tree. This simplifies the implementation of several contract inference rules.

The are four main modules that comprise DF4ABS:

1. *Contract and Constraint Generation*. This is performed in three steps: (1) 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; (2) then, it parses again all classes of the program to generate the initial environment Γ that maps methods to the corresponding method signatures; and (3) it finally parses the AST and, at each node, it applies the contract inference rules in Figs. 9, 10, and 11.
2. *Constraint Solving* is done by a generic semi-unification solver implemented in Java, following the algorithm defined in [20]. When the solver terminates (and no error is found), it produces a substitution that satisfies the input constraints. Applying this substitution to the generated contracts produces the abstract class table and the contract of the main function of the program.
3. *Fixpoint Analysis* uses dynamic structures to store lams of every method contract (because lams become larger and larger as the analysis progresses). At each iteration of the analysis, a number of fresh cog names are created and the states are updated according to what is prescribed by the contract. 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. To detect whether a relation in the fixpoint lam contains a circular dependency, we run Tarjan algorithm [35] for connected components of graphs and we stop the algorithm when a circularity is found.
4. *Abstract model checking* algorithm for deciding the circularity-freedom problem in linear recursive contract class tables performs the following steps. (*i*) *Find (linear) recursive methods*: By parsing the contract class table, we create a graph where nodes are function names, and for every invocation of D.n in the body of C.m, there is an edge from C.m to D.n. Then, a standard depth first search associates with every node a path of (mutual) recursive invocations (the paths starting and ending at that node, if any). The contract class table is linear recursive if every node has at most one associated path. (*ii*) *Computation of the orders*: Given the list of recursive methods, we compute the corresponding mutations. (*iii*) *Evaluation process*: The contract pair corresponding to the main function is evaluated till every recursive function invocation has been unfolded upto twice the corresponding order. (*iv*) *Detection of circularities*: This is performed with the same algorithm of the fixpoint analysis.

As regards the computational complexity, the contract inference system is polynomial time with respect to the length of the program in most of the cases [20]. The fixpoint analysis is exponential in the number of cog names in a contract class table (because lams may double the size at every iteration). However, this exponential effect actually bites in practice. The abstract model checking is linear with respect to the length of the program as far as steps (i) and (ii) are concerned. Step (iv) is linear with respect to the size of the final lam. The critical step is (iii), which may be exponential with respect to the length of the program. Below, there is an overestimation of the computational complexity. Let

$\mathfrak{o}_{max}$	be the largest order of a recursive method contract (without loss of generality, we assume there is no mutual recursion).
m_{max}	be the maximal number of function invocations in a body or in the contract of the main function.

An upper bound to the length of the evaluation till the saturated state is

$$\sum_{0 \le i \le \ell} (2 \times \mathfrak{o}_{max} \times m_{max})^i,$$

where ℓ is the number of methods in the program. Let k_{max} be the maximal number of dependency pairs in a body. Then, the size of the saturated state is $O(k_{max} \times (\mathfrak{o}_{max} \times m_{max})^\ell)$, which is also the computational complexity of the abstract model checking.

7.1 Assessments

We tested DF4ABS on a number of medium-size programs written for benchmarking purposes by core ABS program-

Table 1 Assessments of `DF4ABS`

Program	Lines	DF4ABS/fixpoint		DF4ABS/model-check		DECO	
		Result	Time	Result	Time	Result	Time
`PingPong`	61	✓	0.311	✓	0.046	✓	1.30
`MultiPingPong`	88	D	0.209	D	0.109	D	1.43
`BoundedBuffer`	103	✓	0.126	✓	0.353	✓	1.26
`PeerToPeer`	185	✓	0.320	✓	6.070	✓	1.63
`FAS Module`	2,645	✓	31.88	✓	39.78	✓	4.38

mers and on an industrial case study based on the Fredhopper Access Server (FAS)[5] developed by SDL Fredhopper [34], which provides search and merchandising IT services to e-Commerce companies.

The leftmost two columns of Table 1 reports the experiments: For every program, we display the number of lines, whether the analysis has reported a deadlock (D) or not (✓), the time in seconds required for the analysis. Concerning time, we only report the time of the analysis of DF4ABS (and not the one taken by the inference) when they run on a QuadCore 2.4 GHz and Gentoo (Kernel 3.4.9).

The rightmost column of Table 1 reports the results of another tool that has also been developed for the deadlock analysis of core ABS programs: DECO [12]. This technique integrates a point-to analysis with an analysis returning (an over-approximation of) program points that may be running in parallel. As highlighted by the above table, the three tools return the results as regards deadlock analysis, but are different as regards performance. In particular, the fixpoint and model-checking analysis of DF4ABS are comparable on small-/mid-size programs, and DECO appears less efficient (except for `PeerToPeer`, where our model-checking analysis is quite slow because of the number of dependencies produced by the underlying algorithm). On the `FAS module`, our two analysis are again comparable, while DECO has a better performance (DECO worst case complexity is cubic in the size of the input).

Few remarks about the precision of the techniques follow. DF4ABS/model-check is the most powerful tool we are aware of for linear recursive contract class table. For instance, it correctly detects the deadlock freedom of the method `Math.fact_nc` (previously defined in Fig. 5) while DF4ABS/fixpoint signals a false positive. Similarly, DECO signals a false positive deadlock for the following program, whereas DF4ABS/model-check returns its deadlock freedom.

```
class C implements C {
    Unit m(C c){ C w ;
        w = new cog C() ;
        w!m(this) ;
        c!n(this) ;
        }
    Unit n(C a){ Fut<Unit> x ;
        x = a!q() ;
        x.get ;
        }
    Unit q(){ }
}
{ C a; C b ;
   Fut<Unit> x ;
   a = new cog C() ;
   b = new cog C() ;
   x = a!m(b) ;
}
```

However, DF4ABS/model-check is not defined on nonlinear recursive contract class tables. Nonlinear recursive contract class tables can easily be defined, as shown with the following two contracts:

$$\begin{aligned}
\texttt{C.m} &= [cog : c]\ ()\ \{\langle 0, (\texttt{C!m}\,[cog : c]() \to \_).(c, c') \\
&\qquad + \texttt{C!n}\,[cog : c''](\,[cog : c]) \to \_\rangle\} \to \_ \\
\texttt{C.n} &= [cog : c]\ ([cog : c']) \\
&\qquad \{\langle (\texttt{C!m}\,[cog : c]() \to \_).(c, c'), 0\rangle\} \to \_
\end{aligned}$$

Here, DF4ABS/model-check fails to analyze `C.m`, while DF4ABS/fixpoint and DECO successfully recognize as deadlock-free.[6] We conclude this section with a remark about the proportion between programs with linear recursive contract class tables and those with nonlinear ones. While this proportion is hard to assess, our preliminary analyses strengthen the claim that nonlinear recursive programs are rare. We have parsed the three case studies developed in the European project HATS [34]. The case studies are the FAS module, a Trading System (TS) modeling a supermarket handling sales and a Virtual Office of the Future (VOF) where office workers

[5] Actually, the FAS module has been written in ABS [34], and so, we had to adapt it in order to conform with core ABS restrictions (see Sect. 3). This adaptation just consisted of purely syntactic changes and only took half-day work (see also the comments in [14]).

[6] In [18], we have defined a source-to-source transformation taking nonlinear recursive contract class tables and returning linear recursive ones. This transformation introduces fake cog dependencies that returns a false positive when applying DF4ABS/model-check on the example above.

are enabled to perform their office tasks seamlessly independent of their current location. FAS has 2,645 code-lines, TS has 1,238 code-lines, and VOF has 429 code-lines. In none of them we found a nonlinear recursion in the corresponding contract class table, TS and VOF have, respectively, 2 and 3 linear recursive method contracts (there are recursions in functions on data type values that have nothing to do with locks and control). This substantiates the usefulness of our technique in these programs; the analysis of a wider range of programs is matter of future work.

8 Related works

A preliminary theoretical study was undertaken in [16], 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, method contracts are not pairs of lams, which led it to discard dependencies (thereby causing the analysis, in some cases, to erroneously yield false negatives). This system has been improved in [14] by modeling method contracts as pairs of lams, thus supporting a more precise fixpoint technique. The contract inference system of [14] has been extended in this contribution with the management of aliases of futures and with the dichotomy of present contract and future contract in the inference rules of statements.

The proposals in the literature that statically analyze deadlocks are largely based on (behavioral) types. In [1,2,11,36], 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. Similarly to these techniques, the tool `Java PathFinder` Visser et al. [37] computes a tree of lock orders for every method and searches for mismatches between such orderings. On the contrary, our technique does not compute any ordering of locks during the inference of contracts, 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 Extended Static Checking for Java [10] is an automatic tool for contract-based programming: Annotation is used to specify loop invariants, pre- and post-conditions, and to catch deadlocks. The tool warns the programmer if the annotations cannot be validated. This techniques requires that annotations are explicitly provided by the programmer, while they are inferred in `DF4ABS`.

A well-known deadlock analyzer is TyPiCal, a tool that has been developed for pi-calculus by Kobayashi [21,25–27]. TyPiCal uses a clever technique for deriving inter-channel dependency information and is able to deal with several recursive behaviors and the creation of new channels without committing to any pre-defined order of channel names. Nevertheless, since TyPiCal is based on an inference system, there are recursive behaviors that escape its accuracy. For instance, it returns false positives when recursion create networks with arbitrary numbers of nodes. To illustrate the issue, we consider the following deadlock-free program computing factorial

```
class Math implements Math {
   Int fact(Int n, Int r){
    Math y ;
    Fut<Int> v ;
    if (n == 0) return r ;
        else { y = new cog Math() ;
               v = y!fact(n-1, n*r) ;
               w = v.get ;
               return w ;
          }
    }
}
{
    Math x ; Fut<Int> fut ; Int r ;
    x = new cog Math();
    fut = x!fact(6,1);
    r = fut.get ;
}
```

that is a variation of the method `Math.fact_ng` in Fig. 5. This code is deadlock-free according to `DF4ABS`/model-check; however, its implementation in pi-calculus[7] is not deadlock-free according to TyPiCal. The extension of TyPiCal with a technique similar to the one in Sect. 6, but covering the whole range of lam programs, has been recently defined in [15].

Type-based deadlock analysis has also been studied in [33]. 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, Puntigam and Peter [33] uses a deadlock prevention approach, rather than detection, and no inference system for types is provided.

In [32], the author proposes two approaches for a type- and effect-based deadlock analysis for a concurrent extension of ML. The first approach, like our ones, uses a type and effect

[7] The pi-calculus factorial program is
```
*factorial?(n,(r,s)).
    if n=0 then r?m. s!m else new t in
       (r?m. t!(m*n)) | factorial!(n-1,(t,s))
```
In this code, `factorial` returns the value (on the channel `s`) by *delegating* this task to the recursive invocation, if any. In particular, the initial invocation of `factorial`, which is `r!1 | factorial!(n,(r,s))`, performs a synchronization between `r!1` and the input `r?m` in the continuation of `factorial?(n,(r,s))`. In turn, this may delegate the computation of the factorial to a subsequent synchronization on a new channel `t`. TyPiCal signals a deadlock on the two inputs `r?m` because it fails in connecting the output `t!(m*n)` with them.

inference algorithm, followed by an analysis to verify deadlock freedom. However, their analysis approximates infinite behaviors with a chaotic behavior that non-deterministically acquires and releases locks, thus becoming imprecise. For instance, the previous example should be considered a potential deadlock in their approach. The second approach is an initial result on a technique for reducing deadlock analysis to data race analysis.

Model-theoretical techniques for deadlock analysis have also been investigated. In [3], circular dependencies among processes are detected as erroneous configurations, but dynamic creation of names is not treated. Similarly in [8] (see the Sect. 3.6 below).

Works that specifically tackle the problem of deadlocks for languages with the same concurrency model as that of `core ABS` are the following: West et al. [38] 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 cogs in `core 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. de Boer et al. [8] studied 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. Kerfoot et al. [24] offer 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 the following: (*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 relations with the work by Flores-Montoya et al. [12] have been largely discussed in Sect. 7. Here, we remark that, as regards the design, `DECO` is a monolithic code written in Prolog. On the contrary, `DF4ABS` is a highly modular Java code. 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, in particular, one used in `DECO` (see Sect. 9).

9 Conclusions

We have developed a framework for detecting deadlocks in `core ABS` programs. The technique uses (1) an inference algorithm to extract abstract descriptions of methods, called contracts, (2) an evaluator of contracts, which computes an over-approximated fixpoint semantics and (3) a model-checking algorithm that evaluates contracts by unfolding method invocations.

This study can be extended in several directions. As regards the prototype, the next release will 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 the program is deadlock-free.

`DF4ABS`, being modular, may be integrated with other analysis techniques. In fact, in collaboration with Kobayashi [15], we have recently defined a variant of the model-checking algorithm that has no linearity restriction. For the same reason, another direction of research is to analyze contracts with the point-to analysis technique of `DECO` [12]. We expect that such analyzer will be simpler than `DECO` because, after all, contracts are simpler than `core ABS` programs.

Another direction of research is the application of our inference system to other languages featuring asynchronous method invocation, possibly after removing or adapting or adding rules. One such language that we are currently studying is `ASP` [5]. While we think that our framework and its underlying theory are robust enough to support these applications, we observe that a necessary condition for demonstrating the results of correctness of the framework is that the language has a formal semantics.

Appendix 1: Properties of Sect. 4

The *initial configuration* of a well-typed `core ABS` program is

$$ob(start, \varepsilon, \{[\text{destiny} \mapsto f_{start}, \bar{x} \mapsto \bot] \mid s\}, \varnothing)$$
$$cog(\text{start}, start)$$

where the activity $\{[\text{destiny} \mapsto f_{start}, \bar{x} \mapsto \bot] \mid s\}$ corresponds to the activation of the main function. A *computation* is a sequence of reductions starting at the initial configuration according to the operational semantics. We show in this appendix that such computations keep configurations well-typed; in particular, we show that the sequence of contracts corresponding to the configurations of the computations is in the *later-stage relationship* (see Fig. 27).

9.1 Runtime contracts

In order to type the configurations, we use a *runtime type system*. To this aim we extend the syntax of contracts in Fig. 8 and define *extended futures* F, *extended contracts* that, with an abuse of notation, we still denote $\mathbb{c}$ and *runtime contracts* $\mathbb{k}$ as follows:

$$F ::= f \mid \imath_f$$

$$\mathbb{c} ::= \textit{as in Fig. 8} \mid f \mid f.(c, c') \mid f.(c, c')^{\mathtt{w}} \mid \langle \mathbb{c}, \mathbb{c} \rangle^c$$

$$\mathbb{k} ::= 0 \mid \langle \mathbb{c}, \mathbb{c} \rangle^c_f \mid [\mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}]_f \mid \mathbb{k} \parallel \mathbb{k}$$

As regards F, they are introduced for distinguishing two kind of future names: (1) f that has been used in the contract inference system as a *static time* representation of a future, but is now used as its *runtime* representation; (2) $\imath_f$ now replacing f in its role of *static time* future (it is typically used to reference a future that is not created yet).

As regards $\mathbb{c}$ and $\mathbb{k}$, the extensions are motivated by the fact that, at runtime, the information about contracts is scattered in all the configuration. However, when we plug all the parts to type the whole configuration, we can merge the different information to get a runtime contract $\mathbb{k}'$ such that every contract $\mathbb{c} \in \mathbb{k}'$ does not contain any reference to futures anymore. This merging is done using a set of rewriting rules $\Rightarrow$ defined in Fig. 23 that let one replace the occurrences of runtime futures in runtime contracts $\mathbb{k}$ with the corresponding contract of the future. We write $f \in \mathit{names}(\mathbb{k})$ whenever f occurs in $\mathbb{k}$ not as an index. The substitution $\mathbb{k}[^{\mathbb{c}}/_f]$ replaces the occurrences of f in contracts $\mathbb{c}''$ of $\mathbb{k}$ (by definition of our configurations, in these cases, f can never occur as index in $\mathbb{k}$). It is easy to demonstrate that the merging process always terminates and is confluent for non-recursive contracts, and in the following, we let $(\!|\mathbb{k}|\!)$ be the *normal form* of $\mathbb{k}$ with respect to $\Rightarrow$:

Definition 11 A runtime contract $\mathbb{k}$ is *non-recursive* if:

- all futures $f \in \mathit{names}(\mathbb{k})$ are declared once in $\mathbb{k}$
- all futures $f \in \mathit{names}(\mathbb{k})$ are not recursive, i.e., for all $\langle \mathbb{c}, \mathbb{c}' \rangle^c_f \in \mathbb{k}$, we have $f \notin \mathit{names}(\langle \mathbb{c}, \mathbb{c}' \rangle^c_f)$

9.2 Typing runtime configurations

The typing rules for the runtime configuration are given in Figs. 24, 25 and 26. Except for few rules (in particular, those in Fig. 24 which type the runtime objects of a configuration), all the typing rules have a corresponding one in the contract inference system defined in Sect. 4. Additionally, the typing judgments are identical to the corresponding one in the inference system, with three minor differences:

(i) The typing environment, which now contains a reference to the contract class table and mappings object names to pairs $(\mathtt{C}, \mathbb{r})$, is called Δ;
(ii) the typing rules do not collect constraints;
(iii) the $\mathit{rt\_unsync}(\cdot)$ function on environments Δ is similar to $\mathit{unsync}(\cdot)$ in Sect. 4, except that it now grabs all $\imath_f$ and all futures f' that was created by the current thread f. More precisely,

$$\mathit{rt\_unsync}(\Delta, f) \overset{\mathit{def}}{=} \mathbb{c}_1 \parallel \cdots \parallel \mathbb{c}_n \parallel f_1 \parallel \cdots \parallel f_m$$

where $\{\mathbb{c}_1, \ldots, \mathbb{c}_n\} = \{\mathbb{c}' \mid \exists \imath_f, \mathbb{r} : \ \Delta(\imath_f) = (\mathbb{r}, \mathbb{c}')\}$ and $\{f_1, \ldots, f_m\} = \{f' \mid \Delta(f') = (\mathbb{r}', f)\}$.

$$\frac{f \in \mathit{names}(\mathbb{k})}{\mathbb{k} \| \langle \mathbb{c}, \mathbb{c}' \rangle^c_f \Rightarrow \mathbb{k}[^{\langle \mathbb{c}, \mathbb{c}' \rangle^c}/_f]} \qquad \frac{f \in \mathit{names}(\mathbb{k})}{\mathbb{k} \| [\mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}]_f \Rightarrow \mathbb{k}[^{\mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}}/_f]}$$

Fig. 23 Definition of $\Rightarrow$

(TR-Future-Tick)
$$\frac{\Delta(f) = (c \rightsquigarrow \mathbb{r}, \mathbb{c})^{\checkmark} \qquad \Delta \vdash \mathit{val} : \mathbb{r}}{\Delta \vdash_R \mathit{fut}(f, \mathit{val}) : 0}$$

(TR-Future)
$$\frac{\Delta(f) = (c \rightsquigarrow \mathbb{r}, \mathbb{c})}{\Delta \vdash_R \mathit{fut}(f, \bot) : 0}$$

(TR-Invoc)
$$\frac{\Delta(f) = (c \rightsquigarrow \mathbb{r}', \mathbb{c}) \qquad \Delta \vdash_R \overline{v} = \overline{\mathbb{r}} \qquad \Delta(o) = [\mathit{cog} : c, \overline{x{:}\mathbb{r}}]}{\Delta \vdash_R \mathit{invoc}(o, f, m, \overline{v}) : [\mathtt{C!m}\ [\mathit{cog} : c, \overline{x{:}\mathbb{r}}](\overline{\mathbb{s}}) \to \mathbb{r}']_f}$$

(TR-Object)
$$\frac{\Delta(o) = [\mathit{cog} : c, \overline{x{:}\mathbb{r}}] \qquad \Delta \vdash_R^{c,o} \overline{\mathit{val}} : \overline{\mathbb{r}} \qquad \Delta \vdash_R^{c,o} p : \mathbb{k} \qquad \Delta \vdash_R^{c,o} \overline{p} : \overline{\mathbb{k}}}{\Delta \vdash_R \mathit{ob}(o, [\mathit{cog} \mapsto c; \overline{x \mapsto \mathit{val}}], p, \overline{p}) : \mathbb{k} \| \overline{\mathbb{k}}}$$

(TR-Process)
$$\frac{\Delta \vdash_R^{c,o} \overline{\mathit{val} : \mathbb{x}} \qquad \Delta(f) = (c \rightsquigarrow \mathbb{r}', \overline{f'})^{[\checkmark]} \qquad \Delta[\text{destiny} \mapsto f, \overline{x \mapsto \mathbb{x}}] \vdash_R^{c,o} s : \mathbb{c} \mid \Delta''}{\Delta \vdash_R^{c,o} \{\text{destiny} \mapsto f, \overline{x \mapsto \mathit{val}} \mid s\} : \langle \mathbb{c}, \mathit{rt\_unsync}(\Delta'', f) \rangle^c_f}$$

(TR-Idle)
$$\Delta \vdash_R^{c,o} \mathtt{idle} : 0$$

(TR-Parallel)
$$\frac{\Delta \vdash_R \mathit{cn}_1 : \mathbb{k}_1 \qquad \Delta \vdash_R \mathit{cn}_2 : \mathbb{k}_2}{\Delta \vdash_R \mathit{cn}_1\ \mathit{cn}_2 : \mathbb{k}_1 \parallel \mathbb{k}_2}$$

Fig. 24 Typing rules for runtime configurations

runtime expressions

(TR-Obj)
$$\frac{\Delta(o) = (\mathtt{C}, \mathbb{r})}{\Delta \vdash_R^{c,o} o : \mathbb{r}}$$

(TR-Fut)
$$\frac{\Delta(F) = \mathbb{z}}{\Delta \vdash_R^{c,o} F : \mathbb{z}}$$

(TR-Var)
$$\frac{\Delta(x) = \mathbb{x}}{\Delta \vdash_R^{c,o} x : \mathbb{x}}$$

(TR-Field)
$$\frac{x \notin \mathrm{dom}(\Delta) \qquad \Delta(o.x) = \mathbb{r}}{\Delta \vdash_R^{c,o} x : \mathbb{r}}$$

(TR-Value)
$$\frac{\Delta \vdash_R^{c,o} e : F \qquad \Delta \vdash_R^{c,o} F : (\mathbb{r}, \mathbb{c})^{[\checkmark]}}{\Delta \vdash_R^{c,o} e : \mathbb{r}}$$

(TR-Val)
$$\frac{e \quad \textit{primitive value or arithmetic-and-bool-exp}}{\Delta \vdash_R^{c,o} e : \_}$$

(TR-Pure)
$$\frac{\Delta \vdash_R^{c,o} e : \mathbb{r}}{\Delta \vdash_R^{c,o} e : \mathbb{r}, 0 \mid \Delta}$$

expressions with side-effects

(TR-Get)
$$\frac{\begin{array}{c}\Delta \vdash_R^{c,o} x : \imath_f \qquad \Delta \vdash_R^{c,o} \imath_f : (c' \rightsquigarrow \mathbb{r}', \mathbb{c}) \\ \Delta[\mathit{destiny}] = f \qquad \Delta' = \Delta[\imath_f \mapsto (\mathbb{r}, 0)^{\checkmark}]\end{array}}{\Delta \vdash_R^{c,o} x.\mathtt{get} : \mathbb{r}', \mathbb{c}.(c, c') \parallel \mathit{rt\_unsync}(\Delta', f) \mid \Delta'}$$

(TR-Get-Runtime)
$$\frac{\begin{array}{c}\Delta \vdash_R^{c,o} x : f \qquad \Delta \vdash_R^{c,o} f : (c' \rightsquigarrow \mathbb{r}', \mathbb{c}) \\ \Delta[\mathit{destiny}] = f' \qquad \Delta' = \Delta[f \mapsto (\mathbb{r}, 0)^{\checkmark}]\end{array}}{\Delta \vdash_R^{c,o} x.\mathtt{get} : \mathbb{r}', f.(c, c') \parallel \mathit{rt\_unsync}(\Delta', f') \mid \Delta'}$$

(TR-Get-tick)
$$\frac{\Delta \vdash_R^{c,o} x : F \qquad \Delta \vdash_R^{c,o} F : (c' \rightsquigarrow \mathbb{r}', \mathbb{c})^{\checkmark}}{\Delta \vdash_R^{c,o} x.\mathtt{get} : \mathbb{r}', 0 \mid \Delta}$$

(TR-NewCog)
$$\frac{\Delta \vdash_R^{c,o} \overline{e} : \overline{\mathbb{r}} \qquad \mathit{param}(\mathtt{C}) = \overline{T\ x} \qquad \mathit{fields}(\mathtt{C}) = \overline{T'\ x'} \qquad c' \text{ fresh}}{\Delta \vdash_R^{c,o} \mathtt{new\ cog\ C}(\overline{e}) : [\mathit{cog}{:}c', \overline{x{:}\mathbb{r}}, \overline{x'}{:}\overline{\mathbb{r}'}], 0 \mid \Delta}$$

(TR-New)
$$\frac{\Delta \vdash_R^{c,o} \overline{e} : \overline{\mathbb{r}} \qquad \mathit{param}(\mathtt{C}) = \overline{T\ x} \qquad \mathit{fields}(\mathtt{C}) = \overline{T'\ x'}}{\Delta \vdash_R^{c,o} \mathtt{new\ C}(\overline{e}) : [\mathit{cog}{:}c, \overline{x{:}\mathbb{r}}, \overline{x'}{:}\overline{\mathbb{r}'}], 0 \mid \Delta}$$

(TR-AInvk)
$$\frac{\begin{array}{c}\Delta \vdash_R^{c,o} e : [\mathit{cog}{:}c', \overline{x{:}\mathbb{r}}] \qquad \mathit{class}(\mathit{types}(e)) = \mathtt{C} \qquad \Delta \vdash_R^{c,o} \overline{e} : \overline{\mathbb{s}} \qquad \mathit{fields}(\mathtt{C}), \mathit{param}(\mathtt{C}) = \overline{T\ x} \\ \Delta(\mathtt{C.m}) = \mathbb{r}'(\overline{\mathbb{s}'})\{\langle \mathbb{c}, \mathbb{c}' \rangle\}\mathbb{r}'' \qquad \overline{c'} = \mathit{cog\_names}(\mathbb{r}'') \setminus \mathit{cog\_names}(\mathbb{r}', \overline{\mathbb{s}'}) \qquad \overline{c}, \imath_f \text{ fresh} \qquad \mathbb{s}'' = \mathbb{r}''[^{\overline{c}}/_{\overline{c'}}][^{\mathbb{r}, \overline{\mathbb{s}}}/_{\mathbb{r}', \overline{\mathbb{s}'}}]\end{array}}{\Delta \vdash_R^{c,o} e!\mathtt{m}(\overline{e}) : \imath_f, 0 \mid \Delta[\imath_f \mapsto (c' \rightsquigarrow \mathbb{s}'', \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{s}'')]}$$

(TR-SInvk)
$$\frac{\begin{array}{c}\Delta \vdash_R^{c,o} e : [\mathit{cog}{:}c', \overline{x{:}\mathbb{r}}] \qquad \mathit{class}(\mathit{types}(e)) = \mathtt{C} \qquad \Delta \vdash_R^{c,o} \overline{e} : \overline{\mathbb{s}} \qquad \mathit{fields}(\mathtt{C}), \mathit{param}(\mathtt{C}) = \overline{T\ x} \\ \Delta(\mathtt{C.m}) = \mathbb{r}'(\overline{\mathbb{s}'})\{\langle \mathbb{c}, \mathbb{c}' \rangle\}\mathbb{r}'' \qquad \overline{c'} = \mathit{cog\_names}(\mathbb{r}'') \setminus \mathit{cog\_names}(\mathbb{r}', \overline{\mathbb{s}'}) \qquad \overline{c} \text{ fresh} \qquad \mathbb{s}'' = \mathbb{r}''[^{\overline{c}}/_{\overline{c'}}][^{\mathbb{r}, \overline{\mathbb{s}}}/_{\mathbb{r}', \overline{\mathbb{s}'}}]\end{array}}{\Delta \vdash_R^{c,o} e.\mathtt{m}(\overline{e}) : \mathbb{s}'', \mathtt{C.m}\ \mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{s}'' \parallel \mathit{rt\_unsync}(\Delta) \mid \Delta}$$

Fig. 25 Runtime typing rules for expressions

Finally, few remarks about the auxiliary functions:

- $\mathit{init}(\mathtt{C}, o)$ is supposed to return the init activity of the class C. However, we have assumed that these activity is always empty, see Footnote 2. Therefore, the corresponding contract will be $\langle 0, 0 \rangle$.
- $\mathit{atts}(\mathtt{C}, \overline{v}, o, c)$ returns a substitution provided that $\overline{v}$ have records $\overline{\mathbb{r}}$ and o and c are object and cog identifiers, respectively.
- $\mathit{bind}(o, f, \mathtt{m}, \overline{v'}, \mathtt{C})$ returns the activity corresponding to the method C.m with the parameters $\overline{v'}$ provided that f has type $c \rightsquigarrow \mathbb{r}$ and $\overline{v'}$ have the types $\overline{\mathbb{r}'}$.

Theorem 3 *Let $P = \bar{I}\ \bar{C}\ \{\overline{T\ x}; s\}$ be a* core ABS *program and let $\Gamma \vdash P$:* CCT, $\langle \mathbb{c}, \mathbb{c}' \rangle \rhd \mathcal{U}$. *Let also σ be a substitution satisfying $\mathcal{U}$ and*

$$\Delta = \sigma(\Gamma + \text{CCT}) + \mathit{start} : [\mathit{cog} : \text{start}] + f_{\mathit{start}} : (\text{start} \rightsquigarrow \_, 0)$$

Then

$$\Delta \vdash_R \mathit{ob}(\mathit{start}, \varepsilon, \{l \mid s\}, \emptyset)\ \mathit{cog}(\text{start}, \mathit{start}) : \sigma(\langle \mathbb{c}, \mathbb{c}' \rangle)^{\text{start}}_{f_{\mathit{start}}}$$

where $l = [\mathit{destiny} \mapsto f_{\mathit{start}}, \overline{x \mapsto \bot}]$.

Proof By (TR- Configuration) and (TR- Object) we are reduced to prove:

$$\Delta \vdash_R^{\text{start}, \mathit{start}} \{\text{destiny} \mapsto f_{\mathit{start}}, \overline{x \mapsto \bot} | s\} : \sigma(\langle \mathbb{c}, \mathbb{c}' \rangle)^{\text{start}}_{f_{\mathit{start}}} \tag{3}$$

To this aim, let $\bar{X}$ be the variables used in the inference rule of (T- Program).

To demonstrate (3) we use (TR- Process). Therefore, we need to prove:

$$\Delta[\text{destiny} \mapsto f_{\mathit{start}}, \overline{x \mapsto \sigma(X)}] \vdash_R^{\text{start}, \mathit{start}} s : \sigma(\mathbb{c}) \mid \Delta'$$

with $\mathit{rt\_unsync}(\Delta') = \sigma(\mathbb{c}'')$. This proof is done by a standard induction on s, using a derivation tree identical to the one used for the inference (with the minor exception of replacing the fs used in the inference with corresponding $\imath_f$s). This is omitted because straightforward. □

statements

(TR-Var-Record)
$$\frac{\Delta \vdash^{c,o}_{R} x : \mathbb{x} \qquad \Delta \vdash^{c,o}_{R} z : \mathbb{x}', \mathbb{c} \mid \Delta'}{\Delta \vdash^{c,o}_{R} x = z : \mathbb{c} \mid \Delta'[x \mapsto \mathbb{x}']}$$

(TR-Field-Record)
$$\frac{x \notin \mathrm{dom}(\Delta) \qquad \Delta(\mathtt{this}.x) = \mathbb{r} \qquad \Delta \vdash^{c,o}_{R} z : \mathbb{r}, \mathbb{c} \mid \Delta'}{\Delta \vdash^{c,o}_{R} x = z : \mathbb{c} \mid \Delta'}$$

(TR-Var-Future)
$$\frac{\Delta \vdash^{c,o}_{R} x : F}{\Delta \vdash^{c,o}_{R} x = f : 0 \mid \Delta[x \mapsto f]}$$

(TR-Await)
$$\frac{\begin{array}{c}\Delta \vdash^{c,o}_{R} x : \mathbb{1}_f \qquad \Delta \vdash^{c,o}_{R} \mathbb{1}_f : (c' \rightsquigarrow \mathbb{r}, \mathbb{c}) \\ \Delta[destiny] = f \qquad \Delta' = \Delta[\mathbb{1}_f \mapsto (c' \rightsquigarrow \mathbb{r}, 0)^{\checkmark}]\end{array}}{\Delta \vdash^{c,o}_{R} \mathtt{await}\ x? : \mathbb{c}.(c, c')^{\mathtt{w}} \parallel rt\_unsync(\Delta', f) \mid \Delta'}$$

(TR-Await-Runtime)
$$\frac{\begin{array}{c}\Delta \vdash^{c,o}_{R} x : f \qquad \Delta \vdash^{c,o}_{R} f : (c' \rightsquigarrow \mathbb{r}, \mathbb{c}) \\ \Delta[destiny] = f' \qquad \Delta' = \Delta[f \mapsto (c' \rightsquigarrow \mathbb{r}, 0)^{\checkmark}]\end{array}}{\Delta \vdash^{c,o}_{R} \mathtt{await}\ x? : f.(c, c')^{\mathtt{w}} \parallel rt\_unsync(\Delta', f') \mid \Delta'}$$

(TR-Await-Tick)
$$\frac{\Delta \vdash^{c,o}_{R} x : F \qquad \Delta \vdash^{c,o}_{R} F : (c' \rightsquigarrow \mathbb{r}, \mathbb{c})^{\checkmark}}{\Delta \vdash^{c,o}_{R} \mathtt{await}\ x? : 0 \mid \Delta}$$

(TR-If)
$$\frac{\begin{array}{c}\Delta \vdash^{c,o}_{R} e : \mathtt{Bool} \qquad \Delta \vdash^{c,o}_{R} s_1 : \mathbb{c}_1 \mid \Delta_1 \qquad \Delta \vdash^{c,o}_{R} s_2 : \mathbb{c}_2 \mid \Delta_2 \\ x \in \mathrm{dom}(\Delta) \Longrightarrow \Delta_1(x) = \Delta_2(x) \\ x \in \mathtt{Fut}(\Delta) \Longrightarrow \Delta_1(\Delta_1(x)) = \Delta_2(\Delta_2(x)) \\ \Delta' = \Delta_1 + (\Delta_2 \setminus (\mathrm{dom}(\Delta) \cup \{\Delta_2(x) \mid x \in \mathtt{Fut}(\Delta_2)\}))\end{array}}{\Delta \vdash^{c,o}_{R} \mathtt{if}\ e\ \{\, s_1 \,\}\ \mathtt{else}\ \{\, s_2 \,\} : \mathbb{c}_1 + \mathbb{c}_2 \mid \Delta'}$$

(TR-Skip)
$$\Delta \vdash^{c,o}_{R} \mathtt{skip} : 0 \mid \Delta$$

(TR-Seq)
$$\frac{\begin{array}{c}\Delta \vdash^{c,o}_{R} s_1 : \mathbb{c}_1 \mid \Delta_1 \\ \Delta_1 \vdash^{c,o}_{R} s_2 : \mathbb{c}_2 \mid \Delta_2\end{array}}{\Delta \vdash^{c,o}_{R} s_1 ; s_2 : \mathbb{c}_1 \fatsemi \mathbb{c}_2 \mid \Delta_2}$$

(TR-Return)
$$\frac{\begin{array}{c}\Delta \vdash^{c,o}_{R} e : \mathbb{r} \\ \Delta(\mathrm{destiny}) = f \qquad \Delta(f) = (c \rightsquigarrow \mathbb{r}, \mathbb{c})\end{array}}{\Delta \vdash^{c,o}_{R} \mathtt{return}\ e : 0 \mid \Delta}$$

(TR-Cont)
$$\frac{\Delta(f) = \mathbb{z}}{\Delta \vdash^{c,o}_{R} \mathtt{cont}(f) : 0 \mid \Delta}$$

Fig. 26 Runtime typing rules for statements

Definition 12 A runtime contract $\mathbb{k}$ is *well formed* if it is non-recursive and if futures and method calls in $\mathbb{k}$ are placed as described by the typing rules: i.e., in a sequence $\mathbb{c}_1 \fatsemi \ldots \fatsemi \mathbb{c}_n$, they are present in all $\mathbb{c}_i$, $i_1 \leq i \leq i_k$ with $\mathbb{c}_{i_1}$ being when the method is called, and $\mathbb{c}_{i_k}$ being when the method is synchronized with. Formally, for all $\langle \mathbb{c}, \mathbb{c}' \rangle^{c}_{f} \in \mathbb{k}$, we can derive $\varnothing \vdash \mathbb{c} : \mathbb{c}'$ with the following rules:

$$0 \vdash 0 : 0 \qquad 0 \vdash \mathtt{C.m}\ \mathbb{r}(\overline{\mathbb{r}}) \rightarrow \mathbb{r}' : 0 \qquad \frac{\mathbb{c}' = 0 \lor \mathbb{c}' = f}{\mathbb{c}' \vdash f : f}$$

$$\frac{\begin{array}{c}\mathbb{c} = \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \rightarrow \mathbb{r}' \\ \mathbb{c}' = 0 \lor \mathbb{c}' = \mathbb{c}\end{array}}{\mathbb{c}' \vdash \mathbb{c} : \mathbb{c}} \qquad \frac{\mathbb{c}' = 0 \lor \mathbb{c}' = f}{\mathbb{c}' \vdash f.(c, c')^{[\mathtt{w}]} \vdash 0} \qquad \frac{\begin{array}{c}\mathbb{c} = \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \rightarrow \mathbb{r}' \\ \mathbb{c}' = 0 \lor \mathbb{c}' = \mathbb{c}\end{array}}{\mathbb{c}' \vdash \mathbb{c}.(c, c')^{[\mathtt{w}]} : 0}$$

$$\frac{\mathbb{c}' \vdash \mathbb{c}_1 : \mathbb{c}'' \qquad \mathbb{c}'' \vdash \mathbb{c}_2 : \mathbb{c}'''}{\mathbb{c}' \vdash \mathbb{c}_1 \fatsemi \mathbb{c}_2 : \mathbb{c}'''} \qquad \frac{\mathbb{c}' \vdash \mathbb{c}_1 : \mathbb{c}''}{\mathbb{c}' \vdash \mathbb{c}_1 \fatsemi 0 : \mathbb{c}''}$$

$$\frac{\mathbb{c}' \vdash \mathbb{c}_1 : \mathbb{c}'' \qquad \mathbb{c}' \vdash \mathbb{c}_2 : \mathbb{c}''}{\mathbb{c}' \vdash \mathbb{c}_1 + \mathbb{c}_2 : \mathbb{c}''} \qquad \frac{\mathbb{c}'_1 \vdash \mathbb{c} : \mathbb{c}''_1 \qquad \mathbb{c}'_2 \vdash \mathbb{c}' : \mathbb{c}''_2}{\mathbb{c}'_1 \parallel \mathbb{c}'_2 \vdash \mathbb{c} \parallel \mathbb{c}' : \mathbb{c}''_1 \parallel \mathbb{c}''_2}$$

Lemma 1 *If $\Delta \vdash cn : \mathbb{k}$ is a valid statement, then $\mathbb{k}$ is well formed.*

Proof The result is given by the way $rt\_unsync(\cdot)$ is used in the typing rules. □

In the following theorem, we use the so-called *later-stage relation* $\trianglerighteq$ that has been defined in Fig. 27 on runtime contracts.

We observe that the later-stage relation uses a substitution process that *also performs a pattern matching operation*—therefore, it is partial because the pattern matching may fail. In particular, $[^{\mathbb{s}}/_{\mathbb{r}}]$ (1) extracts the cog names and terms $\mathbb{s}'$ in $\mathbb{s}$ that corresponds to occurrences of cog names and record variables in $\mathbb{r}$ and (2) returns the corresponding substitution.

Theorem 4 (Subject reduction) *Let $\Delta \vdash_R cn : \mathbb{k}$ and $cn \rightarrow cn'$. Then, there exist Δ', $\mathbb{k}'$, and an injective renaming of cog names ι such that*

- *$\Delta' \vdash_R cn' : \mathbb{k}'$ and*
- *$\iota(\mathbb{k}) \trianglerighteq \mathbb{k}'$.*

Proof The proof is a case analysis on the reduction rule used in $cn \rightarrow cn'$, and we assume that the evaluation of an expression $[\![e]\!]_\sigma$ always terminates. We focus on the most interesting cases. We remark that the injective renaming ι is used to identify fresh cog names that are created by the static analysis with fresh cog names that are created by the operational semantics. In fact, the renaming is not the identity only in the case of cog creation (second case below).

- *Skip Statement.*

(Skip)
$$ob(o, a, \{l \mid \mathtt{skip}; s\}, q) \rightarrow ob(o, a, \{l \mid s\}, q)$$

the substitution process

$$
\begin{aligned}
[\text{-}/\_] &\stackrel{def}{=} \varepsilon \\
[{}^{\mathbb{r}}/X] &\stackrel{def}{=} [{}^{\mathbb{r}}/X] \\
[{}^{[cog:c', x_1:\mathbb{r}'_1, \cdots, x_n:\mathbb{r}'_n]}/{}_{[cog:c, x_1:\mathbb{r}_1, \cdots, x_n:\mathbb{r}_n]}] &\stackrel{def}{=} [{}^{c'}/{}_{c}]\,[{}^{\mathbb{r}'_1}/{}_{\mathbb{r}_1}] \cdots [{}^{\mathbb{r}'_n}/{}_{\mathbb{r}_n}] \\
[{}^{c' \rightsquigarrow \mathbb{r}'}/{}_{c \rightsquigarrow \mathbb{r}}] &\stackrel{def}{=} [{}^{c'}/{}_{c}]\,[{}^{\mathbb{r}'}/{}_{\mathbb{r}}]
\end{aligned}
$$

the later-stage relation is the least congruence with respect to runtime contracts that contains the rules

LS-Global
$$\frac{\mathbb{k}_1 \trianglerighteq \mathbb{k}'_1 \qquad \mathbb{k}_2 \trianglerighteq \mathbb{k}'_2}{\mathbb{k}_1 \parallel \mathbb{k}_2 \trianglerighteq \mathbb{k}'_1 \parallel \mathbb{k}'_2}$$

LS-Bind
$$\frac{\begin{array}{cc}\Delta(\mathtt{C.m}) = \mathbb{r}_{\mathtt{this}}\ (\overline{\mathbb{r}_{\mathtt{this}}})\ \{\langle \mathbb{c}, \mathbb{c}' \rangle\}\ \mathbb{r}'_{\mathtt{this}} & \overline{c} = fn(\langle \mathbb{c}, \mathbb{c}' \rangle) \setminus fn(\mathbb{r}_{\mathtt{this}}, \overline{\mathbb{r}_{\mathtt{this}}}, \mathbb{r}'_{\mathtt{this}}) \\ \mathbb{r}_{\mathtt{p}} = [cog : c, \overline{x{:}\mathbb{r}}] & \overline{c'} \cap fn(\mathbb{r}_{\mathtt{p}}, \overline{\mathbb{r}_{\mathtt{p}}}, \mathbb{r}'_{\mathtt{p}}) = \varnothing\end{array}}{[\mathtt{C!m}\ \mathbb{r}_{\mathtt{p}}(\overline{\mathbb{r}_{\mathtt{p}}}) \to \mathbb{r}'_{\mathtt{p}}]_f \trianglerighteq \langle \mathbb{c}, \mathbb{c}' \rangle^c_f [{}^{\overline{c'}}/{}_{\overline{c}}][{}^{\mathbb{r}_{\mathtt{p}}, \overline{\mathbb{r}_{\mathtt{p}}}, \mathbb{r}'_{\mathtt{p}}}/{}_{\mathbb{r}_{\mathtt{this}}, \overline{\mathbb{r}_{\mathtt{this}}}, \mathbb{r}'_{\mathtt{this}}}]}$$

LS-AInvk
$$\frac{f' \in fn(\langle \mathbb{c}, \mathbb{c}' \rangle)}{\langle \mathbb{c}, \mathbb{c}' \rangle^c_f [{}^{\mathtt{C!m}\ \mathbb{r}_{\mathtt{p}}(\overline{\mathbb{r}_{\mathtt{p}}}) \to \mathbb{r}'_{\mathtt{p}}}/{}_{f'}] \trianglerighteq \langle \mathbb{c}, \mathbb{c}' \rangle^c_f \parallel [\mathtt{C!m}\ \mathbb{r}_{\mathtt{p}}(\overline{\mathbb{r}_{\mathtt{p}}}) \to \mathbb{r}'_{\mathtt{p}}]_{f'}}$$

LS-SInvk
$$\frac{f' \in fn(\langle \mathbb{c}, \mathbb{c}' \rangle) \qquad \mathbb{r}_{\mathtt{p}} = [cog : c, \overline{x{:}\mathbb{r}}]}{\langle (\mathtt{C.m}\ \mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{r}' \parallel \mathbb{c}) \fatsemi \mathbb{c}', \mathbb{c}'' \rangle^c_f \trianglerighteq \langle (f'.(c,c)^{\mathtt{w}} \parallel \mathbb{c}) \fatsemi \mathbb{c}', \mathbb{c}'' \rangle^c_f \parallel [\mathtt{C!m}\ \mathbb{r}_{\mathtt{p}}(\overline{\mathbb{r}_{\mathtt{p}}}) \to \mathbb{r}'_{\mathtt{p}}]_{f'}}$$

LS-RSInvk
$$\frac{f' \in fn(\langle \mathbb{c}, \mathbb{c}' \rangle) \qquad \mathbb{r}_{\mathtt{p}} = [cog : c', \overline{x{:}\mathbb{r}}] \qquad c' \neq c}{\langle (\mathtt{C.m}\ \mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{r}' \parallel \mathbb{c}) \fatsemi \mathbb{c}', \mathbb{c}'' \rangle^c_f \trianglerighteq \langle (f'.(c,c') \parallel \mathbb{c}) \fatsemi \mathbb{c}', \mathbb{c}'' \rangle^c_f \parallel [\mathtt{C!m}\ \mathbb{r}_{\mathtt{p}}(\overline{\mathbb{r}_{\mathtt{p}}}) \to \mathbb{r}'_{\mathtt{p}}]_{f'}}$$

LS-DepNull
$$\langle \mathbb{c}, \mathbb{c}' \rangle^c_f \parallel \langle 0, 0 \rangle^{c'}_{f'} \trianglerighteq \langle \mathbb{c}[{}^0/{}_{f'}], \mathbb{c}'[{}^0/{}_{f'}] \rangle^c_f \parallel \langle 0, 0 \rangle^{c'}_{f'}$$

LS-Fut
$f \trianglerighteq 0$

LS-Empty
$0.(c,c')^{[\mathtt{w}]} \trianglerighteq 0$

LS-Delete
$0 \fatsemi \mathbb{c} \trianglerighteq \mathbb{c}$

LS-Plus
$\mathbb{c}_1 + \mathbb{c}_2 \trianglerighteq \mathbb{c}_i$

Fig. 27 Later-stage relation

By (TR-Object), (TR-Process), (TR-Seq) and (TR-Skip), there exists Δ'' and $\mathbb{c}$ such that $\Delta'' \vdash^{c,o}_R \mathtt{skip}; s : 0 \fatsemi \mathbb{c} \mid \Delta''$. It is easy to see that $\Delta'' \vdash^{c,o}_R s : \mathbb{c} \mid \Delta''$. Moreover, by (LS-Delete), we have $0 \fatsemi \mathbb{c} \trianglerighteq_{\mathtt{cog}(o)} \mathbb{c}$ which proves that $\mathbb{k} \trianglerighteq \mathbb{k}'$.

– *Object creation.*

(New-Object)
$$\frac{o' = \text{fresh}(\mathtt{C}) \quad p = \text{init}(\mathtt{C}, o') \quad a' = \text{atts}(\mathtt{C}, [\![\overline{e}]\!]_{(a+l)}, c)}{\begin{array}{l} ob(o, a, \{l \mid x = \mathtt{new}\ \mathtt{C}(\overline{e}); s\}, q)\ cog(c, o) \\ \to ob(o, a, \{l \mid x = o'; s\}, q)\ cog(c, o) \quad ob(o', a', \text{idle}, \{p\}) \end{array}}$$

By (TR-Object) and (TR-Process), there exists Δ'' that extends Δ such that $\Delta'' \vdash^{c,o}_R \mathtt{new}\ \mathtt{C}(\overline{e}) : \mathbb{r}, 0 \mid \Delta''$. Let $\Delta' = \Delta[o' \mapsto \mathbb{r}]$. The theorem follows by the assumption that p is empty (see Footnote 2).

– *Cog creation.*

(New-Cog-Object)
$$\frac{\begin{array}{c} c' = \text{fresh}(\,) \quad o' = \text{fresh}(\mathtt{C}) \quad p = \text{init}(\mathtt{C}, o') \\ a' = \text{atts}(\mathtt{C}, [\![\overline{e}]\!]_{(a+l)}, c') \end{array}}{\begin{array}{l} ob(o, a, \{l \mid x = \mathtt{new\ cog}\ \mathtt{C}(\overline{e}); s\}, q) \\ \to ob(o, a, \{l \mid x = o'; s\}, q) \quad ob(o', a', p, \varnothing) \quad cog(c', o') \end{array}}$$

By (TR-Object) and (TR-Process), there exists Δ'' that extends Δ such that $\Delta'' \vdash^{c,o}_R \mathtt{new}\ \mathtt{C}(\overline{e}) : [cog : c'', x \mathrel{\bar{:}} \mathbb{r}], 0 \mid \Delta''$ for some c'' and records $\bar{\mathbb{r}}$. Let $\Delta' = \Delta[o' \mapsto [cog : c', x \mathrel{\bar{:}} \mathbb{r}], c' \mapsto cog]$ and $\imath(c'') = c'$, where $\imath$ is an injective renaming on cog names. The theorem follows by the assumption that p is empty (see Footnote 2).

– *Asynchronous calls.*

(Async-Call)
$$\frac{o' = [\![e]\!]_{(a+l)} \quad \overline{v} = [\![\overline{e}]\!]_{(a+l)} \quad f = \text{fresh}(\,)}{\begin{array}{l} ob(o, a, \{l \mid x = e\,!\,\mathtt{m}(\overline{e}); s\}, q) \\ \to ob(o, a, \{l \mid x = f; s\}, q)\ invoc(o', f, \mathtt{m}, \overline{v})\ fut(f, \bot) \end{array}}$$

By (TR-Object) and (TR-Process), there exist $\bar{\mathbb{r}}$, Δ'_1, $\mathbb{c}$ and $\mathbb{k}''$ such that (let $f' = l(\text{destiny})$)

– $\mathbb{k} = \langle \mathbb{c}, rt\_unsync(\Delta'_1, f') \rangle^{f'}_{\mathtt{cog}(o)} \| \mathbb{k}''$
– $\Delta \vdash^{c,o}_R \bar{v} : \bar{\mathbb{x}}$ (with $l = [y \mathrel{\bar{\mapsto}} v]$)
– $\Delta \vdash^{c,o}_R q : \mathbb{k}''$
– $\Delta[y \mathrel{\bar{\mapsto}} \mathbb{r}] \vdash^{c,o}_R x = e!m(\overline{e}); s : \mathbb{c} \mid \Delta'_1$

Let $\Delta_1 = \Delta[y \mathrel{\bar{\mapsto}} \mathbb{r}]$: by either (TR-Var-Record) or (TR-Field-Record) and (TR-AInvk), there exist $\mathbb{r} = c' \rightsquigarrow \mathbb{r}'$ (where c' is the cog of the record of e), $\mathbb{1}_f$ and $\mathbb{c}_{\mathbb{1}_f}$ such that $\Delta_1 \vdash^{c,o}_R e!m(\overline{e}) : \mathbb{1}_f, 0 \mid \Delta_1[\mathbb{1}_f \mapsto (\mathbb{r}, \mathbb{c}_{\mathbb{1}_f})]$.
By construction of the type system (in particular, the rules (TR-Get)* and (TR-Await)*), there exists a term

t such that $\mathbb{c} = t[^{\mathbb{c}_{1f}}/_{1_f}]$ and such that $\Delta_1[f \mapsto (\mathbb{r}, f')] \vdash_R^{c,o} x = f; s : t[^f/_{1_f}] \mid \Delta'_2$ (with $\Delta'_2 \triangleq \Delta'_1 \setminus \{1_f\}[f \mapsto (\mathbb{r}, f')^{[\checkmark]}]$ and $[\checkmark] = \checkmark$ iff $\Delta'_1(1_f)$ is checked). By construction of the *rt_unsync* function, there exist a term t' such that $rt\_unsync(\Delta'_1) = t'[^{\mathbb{c}_{1f}}/_{1_f}]$ and $rt\_unsync(\Delta'_2) = t'[^f/_{1_f}]$.
Finally, if we note $\Delta' \triangleq \Delta[f \mapsto (\mathbb{r}, f')]$, we can type the invocation message with $[\mathbb{c}_{1_f}]_f$ (as c' is the cog of the record of `this` in $\mathbb{c}'$), we have

- $\Delta' \vdash_R cn' : \langle t[^f/_{1_f}], t'[^f/_{1_f}]\rangle_c^{f'} \parallel [\mathbb{c}_{1_f}]_f \parallel \mathbb{k}''$
- the rule (LS- AInvk) gives us that

$$\mathbb{k} \trianglerighteq \langle t[^f/_{1_f}], t'[^f/_{1_f}]\rangle_c^{f'} \parallel [\mathbb{c}_{1_f}]_f \parallel \mathbb{k}''$$

– *Method instantiations.*

(Bind- Mtd)
$$\frac{\{l \mid s\} = \text{bind}(o, f, \mathtt{m}, \overline{v}, \text{class}(o))}{ob(o, a, p, q)\ invoc(o, f, \mathtt{m}, \overline{v}) \rightarrow ob(o, a, p, q \cup \{l \mid s\})}$$

By assumption and rules (TR- Parallel) and (R- Invoc) we have $\Delta(o) = (\mathtt{C}, \mathbb{r})$, $\Delta(f) = (c \rightsquigarrow \mathbb{r}', 0)$, $c = \mathtt{cog}(\mathbb{r})$ and $\mathbb{k} = [C!\mathtt{m}\ \mathbb{r}(\overline{\mathbb{r}}) \rightarrow \mathbb{r}']_f \parallel \mathbb{k}'$ with $\Delta \vdash_R invoc(o, f, m, \overline{v}) : [\mathtt{C}!\mathtt{m}\ \mathbb{r}(\overline{\mathbb{r}}) \rightarrow \mathbb{r}']_f$ and $\Delta \vdash_R ob(o, a, p, q) : \mathbb{k}'$. Let $\overline{x}$ be the formal parameters of $\mathtt{m}$ in $\mathtt{C}$. The auxiliary function $bind(o, f, m, \overline{v}, \mathtt{C})$ returns a process $\{[\text{destiny} \mapsto f, \overline{x} \mapsto \overline{v}] \mid s\}$. It is possible to demonstrate that $\Delta \vdash_R^{c,o} \{l[\text{destiny} \mapsto f, \overline{x} \mapsto \overline{v}] | s\} : \langle \mathbb{c}_{\mathtt{m}}, \mathbb{c}'_{\mathtt{m}}\rangle_c^f$, where $\Delta(\mathtt{C.m}) = \mathbb{s}(\bar{\mathbb{s}})\{\langle \mathbb{c}_0, \mathbb{c}'_0\rangle\}\mathbb{s}'$ and $\mathbb{c}_{\mathtt{m}} = \mathbb{c}_0[^{\bar{c}}/_{\bar{c}'}][^{\mathbb{r}, \bar{\mathbb{r}}}/_{\mathbb{s}, \bar{\mathbb{s}}}]$ and $\bar{c}' \in \mathbb{s}' \setminus (\mathbb{s} \cup \bar{\mathbb{s}})$ with $\bar{c}$ fresh and $\mathbb{c}'_{\mathtt{m}} = \mathbb{c}'_0[^{\bar{c}}/_{\bar{c}'}][^{\mathbb{r}, \bar{\mathbb{r}}}/_{\mathbb{s}, \bar{\mathbb{s}}}]$. By rules (TR- Process) and (TR- Object), it follows that $\Delta \vdash_R ob(o, a, p, q \cup \{bind(o, f, m, \overline{v}, \mathtt{C})\}) : \mathbb{k}' \| \langle \mathbb{c}_{\mathtt{m}}, \mathbb{c}'_{\mathtt{m}}\rangle_c^f$. Moreover, by applying the rule (LS- Bind), we have that $[\mathtt{C}!\mathtt{m}\ \mathbb{r}(\overline{\mathbb{r}}) \rightarrow \mathbb{r}']_f \trianglerighteq \langle \mathbb{c}_{\mathtt{m}}, \mathbb{c}'_{\mathtt{m}}\rangle_c^f$ which implies with the rule (LS- Global) that $\mathbb{k} \trianglerighteq \mathbb{k}'$.

– *Getting the value of a future.*

(Read- Fut)
$$\frac{f = [\![e]\!]_{(a+l)} \quad v \neq \bot}{\begin{array}{l} ob(o, a, \{l \mid x = e.\mathtt{get}; s\}, q)\ fut(f, v) \rightarrow \\ \quad ob(o, a, \{l \mid x = v; s\}, q)\ fut(f, v)\end{array}}$$

By assumption and rules (TR- Parallel), (TR- Object) and (TR- Future- Tick), there exists Δ'', $\mathbb{c}$, $\mathbb{k}''$ such that (let $f' = l\,[\text{destiny}]$)

- $\Delta \vdash_R^{\mathtt{cog}(o),o} \{l | x = e.\mathtt{get}; s\} :$
  $\langle \mathbb{c}, rt\_unsync(\Delta'', f')\rangle_{\mathtt{cog}(o)}^{f'}$
- $\Delta \vdash_R^{\mathtt{cog}(o),o} q : \mathbb{k}''$
- $\Delta \vdash_R fut(f, v) : 0$
- $\mathbb{k} = \langle \mathbb{c}, rt\_unsync(\Delta'', f')\rangle_{\mathtt{cog}(o)}^{f'} \parallel \mathbb{k}''$, and
- $[\![e]\!]_{a \circ l} = f$.

Moreover, as $fut(f, v)$ is typable and contains a value, we know that $\mathbb{c} = 0 \,⨟\, \mathbb{c}'$ (e.`get` has contract 0). With the rule (TR- Pure), have that $\Delta \vdash_R^{\mathtt{cog}(o),o} \{l | x = v; s\} : \langle \mathbb{c}, rt\_unsync(\Delta'', f')\rangle_{\mathtt{cog}(o)}^{f'}$, and with $\mathbb{k}' = \mathbb{k}$, we have the result.

– *Remote synchronous call.* Similar to the cases of asynchronous call with a `get`-synchronization. The result follows, in particular, from rule (LS- RSInvk) of Fig. 27.

– *Cog-local synchronous call.* Similar to case of asynchronous call. The result follows, in particular, from rules (LS- SimpleNull) of Fig. 27 and from the Definition of $\mathtt{C.m}\ \mathbb{r}(\bar{\mathbb{r}}) \rightarrow \mathbb{s}$.

– *Local Assignment.*

(Assign- Local)
$$\frac{x \in \text{dom}(l) \quad v = [\![e]\!]_{(a+l)}}{\begin{array}{l} ob(o, a, \{l \mid x = e; s\}, q) \\ \rightarrow ob(o, a, \{l[x \mapsto v] \mid s\}, q)\end{array}}$$

By assumption and rules (TR- Object), (TR- Process), (TR- Seq), (TR- Var- Record) and (TR- Pure), there exists Δ'', $\mathbb{c}$, $\mathbb{k}''$ such that (we note Δ_1 for $\Delta[\overline{\mathtt{y} : \mathbb{x}}]$ and f for l [destiny])

- $\Delta \vdash_R^{\mathtt{cog}(o),o} \{l | x = e; s\} :$
  $\langle 0 \,⨟\, \mathbb{c}, rt\_unsync(\Delta'', f)\rangle_{\mathtt{cog}(o)}^f$
- $\Delta \vdash_R^{\mathtt{cog}(o),o} q : \mathbb{k}''$
- $\mathbb{k} = \langle \mathbb{c}, rt\_unsync(\Delta'', f)\rangle_{\mathtt{cog}(o)}^f \parallel \mathbb{k}''$, and
- $[\![e]\!]_{a \circ l} = v$.

We have

$$\Delta \vdash_R^{\mathtt{cog}(o),o} \{l[x \mapsto [\![e]\!]_{(a+l)}] | s\} : \\ \langle \mathbb{c}, rt\_unsync(\Delta'', f)\rangle_{\mathtt{cog}(o)}^f$$

which gives us the result with
$\mathbb{k}' = \langle \mathbb{c}, rt\_unsync(\Delta'', f)\rangle_f^c cpar \mathbb{k}'$. □

Appendix 2: Properties of Sect. 5

In this section, we will prove that the statements given in Sect. 5 are correct, i.e., that the fixpoint analysis does detect deadlocks. To prove that statement, we first need to define the dependencies generated by the runtime contract of a running program. Then, our proof works in three steps: (1) First, we show that our analysis (performed at static time) contains all the dependencies of the runtime contract of the program; (2) second, the dependencies in a program at runtime are contained in the dependencies of its runtime contract; and (3) finally, when cn (typed with $\mathbb{k}$) reduces to cn' (typed with $\mathbb{k}'$), we prove that the dependencies of $\mathbb{k}'$ are contained in $\mathbb{k}$.

Basically, we prove that the following diagram holds:

$$\begin{tikzcd}
P \arrow[rrr] \arrow[d, dashed] & & & cn_1 \arrow[dl, dashed] \arrow[dr, dashed] \arrow[rrr, no head] & & & \cdots \arrow[rrr] & & & cn_n \arrow[dl, dashed] \arrow[dr, dashed] & \\
\langle \mathbb{c}, \mathbb{c}' \rangle \arrow[d] & & \Bbbk_1 \arrow[d] & & A_1 \arrow[dl, phantom, sloped, "\Supset"] & & \cdots & & \Bbbk_n \arrow[d] & & A_n \arrow[dl, phantom, sloped, "\Supset"] \\
\langle \mathcal{L}, \mathcal{L}' \rangle & \Supset & \langle \mathcal{L}_1, \mathcal{L}'_1 \rangle & & \Supset & & \cdots & \Supset & \langle \mathcal{L}_n, \mathcal{L}'_n \rangle & &
\end{tikzcd}$$

Hence, the analysis $\langle \mathcal{L}, \mathcal{L}' \rangle$ contains all the dependencies A_i that the program can have at runtime, and thus, if the program has a deadlock, the analysis would have a circularity.

In the following, we introduce how we compute the dependencies of a runtime contract. This computation is difficult in general, but in case the runtime contract is as we constructed it in the subject-reduction theorem, then the definition is very simple. First, let say that a contract $\mathbb{c}$ that does not contain any future is *closed*. It is clear that we can compute $\mathbb{c}(\text{ACT}_{[n]})$ when $\mathbb{c}$ is closed.

Proposition 5 *Let $\Delta \vdash cn : \Bbbk$ be a typing derivation constructed as in the proof of Theorem 4. Then, $\Bbbk$ is well formed and $(\!|\Bbbk|\!) = \langle \mathbb{c}, \mathbb{c}' \rangle^{\text{start}}_{f_{start}}$ where $\mathbb{c}$ and $\mathbb{c}'$ are closed.*

Proof The first property is already stated in Lemma 1. The second property comes from the fact that when we create a new future f (in the *Asynchronous calls* case for instance), we map it in Δ' to its father process, which will then reference f because of the $rt\_unsync(\cdot)$ function. Hence, if we consider the relation of which future references which other future in $\Bbbk$, we get a dependency graph in the shape of a directed tree, where the root is f_{start}. So, $(\!|\Bbbk|\!)$ reduces to a simple pair of contract of the form $\langle \mathbb{c}, \mathbb{c}' \rangle^{\text{start}}_{f_{start}}$ where $\mathbb{c}$ and $\mathbb{c}'$ are closed. □

In the following, we will suppose that all runtime contracts $\Bbbk$ come from a type derivation constructed as in Theorem 4.

Definition 13 The *semantics* of a closed runtime pair (unique upto remaining of cog names) for the saturation at i, noted $[\![\langle \mathbb{c}, \mathbb{c}' \rangle^{c}_{f}]\!]_n$, is defined as $[\![\langle \mathbb{c}, \mathbb{c}' \rangle^{c}_{f}]\!]_n = (\mathbb{c}(\text{ACT}_{[n]})_c) \fatsemi (\mathbb{c}'(\text{ACT}_{[n]})_c)$. We extend that definition for any runtime contract with $[\![\Bbbk]\!]_n \triangleq [\![(\!|\Bbbk|\!)]\!]_n$.

Now we can compute the dependencies of a runtime contract, and we can prove our first property: The analysis performed at static time contains all the dependencies of the initial runtime contract of the program (note that $\sigma(\mathbb{c})(\text{ACT}_{[n]}) \fatsemi \sigma(\mathbb{c}')(\text{ACT}_{[n]})$ is the analysis performed at static time, and $[\![\sigma(\langle \mathbb{c}, \mathbb{c}' \rangle^{\text{start}}_{f_{start}})]\!]_n$ is the set of dependencies of the initial runtime contract of the program):

Proposition 6 *Let $P = \bar{I}\ \bar{C}\ \{\overline{T\ x}; s\}$ be a* `core ABS` *program and let $\Gamma \vdash P : \text{CCT}$, $\langle \mathbb{c}, \mathbb{c}' \rangle \rhd \mathcal{U}$. Let also σ be a substitution satisfying $\mathcal{U}$. Then, we have that $[\![\sigma(\langle \mathbb{c}, \mathbb{c}' \rangle^{\text{start}}_{f_{start}})]\!]_n \Subset \sigma(\mathbb{c})(\text{ACT}_{[n]}) \fatsemi \sigma(\mathbb{c}')(\text{ACT}_{[n]})$.*

Proof The result is direct with an induction on $\mathbb{c}$ and $\mathbb{c}'$, and with the fact that $+$, $\fatsemi$ and $\|$ are monotone with respect to $\Subset$. □

We now prove the second property: All the dependencies of a program at a given time is included in the dependencies generated from its contract.

Proposition 7 *Let suppose $\Delta \vdash_R cn : \Bbbk$ and let A be the set of dependencies of cn. Then, with $[\![\Bbbk]\!]_n = \langle \mathcal{L}, \mathcal{L}' \rangle$, we have $A \subset \mathcal{L}$.*

Proof By Definition 2, if cn has a dependency (c, c'), then there exist $cn_1 = ob(o, a, \{l|x = e.\texttt{get}; s\}, q) \in cn$, $cn_2 = fut(f, \bot) \in cn$ and $cn_3 = ob(o', a', p', q') \in cn$ such that $[\![e]\!]_{(a+l)} = l'(\text{destiny}) = f$, $\{l' \mid s'\} \in p' \cup q'$ and $a(cog) = c$ and $a'(cog) = c'$. By runtime typing rules (TR-Object), (TR-Process), (TR-Seq) and (TR-Get-Runtime), the contract of cn_1 is

$$\langle f.(c, c') \fatsemi \mathbb{c}_s, \mathbb{c}'_s \rangle^{a(\text{cog})}_{l(destiny)} \parallel \Bbbk_q$$

we indeed know that the dependency in the contract is toward c' because of (TR-Invoc) or (TR-Process). Hence, $\Bbbk = \langle f.(c, c') \fatsemi \mathbb{c}_s, \mathbb{c}'_s \rangle^{a(\text{cog})}_{l(destiny)} \parallel \Bbbk'$. It follows, with the lam transformation rule (L-GAInvk), that (c, c') is in $\mathcal{L}$. □

Proposition 8 *Given two runtime contracts $\Bbbk$ and $\Bbbk'$ with $\Bbbk \trianglerighteq \Bbbk'$, we have that $[\![\Bbbk']\!]_n \Subset [\![\Bbbk]\!]_n$.*

Proof We refer to the rules (LS-*) of the later-stage relation defined in Fig. 27 and to the lam transformation rules (L-*) defined in Fig. 16. The result is clear for the rules (LS-Global), (LS-Fut), (LS-Empty), (LS-Delete) and (LS-Plus). The result for the rule (LS-Bind) is a consequence of (L-AInvk). The result for the rule (LS-AInvk) is a consequence of the definition of $\Rightarrow$. The result for the rule (LS-SInvk) is a consequence of the definition of $\Rightarrow$ and (L-SInvk). The result for the rule (LS-RSInvk) is a consequence of the definition of $\Rightarrow$ and (L-RSInvk). Finally, the result for the rule (LS-DepNull) is a consequence of the definition of $\Rightarrow$. □

We can finally conclude by putting all these results together.

Theorem 5 *If a program P has a deadlock at runtime, then its* abstract semantics saturated at n *contains a circle.*

Proof This property is a direct consequence of Propositions 6, 7 and 8. □

Appendix 3: Properties of Sect. 6

The next theorem states the correctness of our model-checking technique.

Below we write $\llbracket cn \rrbracket_{[n]} = (\llbracket \langle \mathbb{cp}_n, \mathbb{cp}'_n \rangle_{\text{start}} \rrbracket)^{\flat}$, if $\Delta \vdash_R cn : \langle \mathbb{cp}_1, \mathbb{cp}'_1 \rangle$ and n is the order of $\langle \mathbb{cp}_1, \mathbb{cp}'_1 \rangle_{\text{start}}$.

Theorem 6 *Let* $(\mathrm{CT}, \{\overline{T\ x\ ;\ s}\}, \mathrm{CCT})$ *be a* `core ABS` *program and cn be a configuration of its operational semantics.*

1. *If cn has a circularity, then a circularity occurs in* $\llbracket cn \rrbracket_{[n]}$*;*
2. *if* $cn \to cn'$ *and* $\llbracket cn' \rrbracket_{[n]}$ *has a circularity, then a circularity is already present in* $\llbracket cn \rrbracket_{[n]}$*;*
3. *let* $\imath$ *be an injective renaming of cog names;* $\llbracket cn \rrbracket_{[n]}$ *has a circularity if and only if* $\llbracket \imath(cn) \rrbracket_{[n]}$ *has a circularity.*

Proof To demonstrate item 1, let

$$\llbracket cn \rrbracket_{[n]} = (\llbracket \langle \mathbb{cp}_n, \mathbb{cp}'_n \rangle_{\text{start}} \rrbracket)^{\flat}.$$

We prove that every dependencies occurring in cn is also contained in one state of $(\llbracket \langle \mathbb{cp}_n, \mathbb{cp}'_n \rangle_{\text{start}} \rrbracket)^{\flat}$. By Definition 2, if cn has a dependency (c, c'), then it contains $cn'' = ob(o, a, \{l | x = e.\mathtt{get}; s\}, q) \;\; fut(f, \bot)$, where $f = \llbracket e \rrbracket_{(a+l)}$, $a(cog) = c$ and there is $ob(o', a', \{l' | s'\}, q') \in cn$ such that $a'(cog) = c'$ and $l'(\text{destiny}) = f$. By the typing rules, the contract of cn' is $f.(c, c') ⨾ \mathbb{c}_s$, where, by typing rule (T- Configurations), f is actually replaced by a $\mathtt{C!m}\,\mathbb{r}(\bar{\mathbb{s}}) \to \mathbb{s}$ produced by a concurrent *invoc* configuration, or by the contract pair $\langle \mathbb{c}_{\mathtt{m}}, \mathbb{c}'_{\mathtt{m}} \rangle$ corresponding to the method body.

As a consequence $\llbracket cn'' \rrbracket_{[n]} = (\llbracket \mathfrak{C}[\langle \mathbb{c}'' \& (c, c'), \mathbb{c}''' \rangle_c]_{c''} \rrbracket)^{\flat}$.

Let $\llbracket ob(o', a', \{l' | s'\}, q') \rrbracket_{[n]} = (\llbracket \mathfrak{C}'[\langle \mathbb{c}_{\mathtt{m}}, \mathbb{c}'_{\mathtt{m}} \rangle_c]_{c''} \rrbracket)^{\flat}$, with $\llbracket e \rrbracket_{(a+l)} = l'(\text{destiny})$, then

$$\begin{aligned}&\llbracket ob(o', a', \{l' | s'\}, q') \;\; cn'' \rrbracket_{[n]} \\ &\quad = (\llbracket \mathfrak{C}[\langle \mathbb{c}'' \& (c, c'), \mathbb{c}''' \rangle_c]_{c''} \parallel \mathfrak{C}'[\langle \mathbb{c}_{\mathtt{m}}, \mathbb{c}'_{\mathtt{m}} \rangle_{c'}]_{c'''} \rrbracket)^{\flat}.\end{aligned}$$

In general, if k dependencies occur in a state cn, then there is $cn'' \subseteq cn$ that collects all the tasks manifesting the dependencies.

$$\begin{aligned}&\llbracket cn'' \rrbracket_{[n]} \\ &\quad = (\llbracket \mathfrak{C}_1[\langle \mathbb{c}''_1 \& (c_1, c'_1), \mathbb{c}'''_1 \rangle_{c_1}]_{c''_1} \parallel \mathfrak{C}'_1[\langle \mathbb{c}_{\mathtt{m}_1}, \mathbb{c}'_{\mathtt{m}_1} \rangle_{c'_1}]_{c'''_1} \rrbracket)^{\flat} \\ &\qquad \parallel \cdots \parallel (\llbracket \mathfrak{C}_k[\langle \mathbb{c}''_k \& (c_k, c'_k), \mathbb{c}'''_k \rangle_{c_k}]_{c''_k} \\ &\qquad \parallel \mathfrak{C}'_k[\langle \mathbb{c}_{\mathtt{m}_k}, \mathbb{c}'_{\mathtt{m}_k} \rangle_{c'_k}]_{c'''_k} \rrbracket)^{\flat}\end{aligned}$$

By definition of $\parallel$ composition in Sect. 5, the initial state contains all the above pairs (c_i, c'_i).

Let us prove the item 2. We show that the transition $cn \longrightarrow cn'$ does not produce new dependencies. That is, the set of dependencies in the states of $\llbracket cn' \rrbracket_{[n]}$ is equal to or smaller than the set of dependencies in the states of $\llbracket cn \rrbracket_{[n]}$.

By Theorem 4, if $\Delta \vdash_R cn : \mathbb{k}$, then $\Delta' \vdash_R cn' : \mathbb{k}'$, with $\mathbb{k} \trianglerighteq \mathbb{k}'$. We refer to the rules (LS- *) of the later-stage relation defined in Fig. 27 and to the contract reduction rules (Red- *) defined in Fig. 19. The result is clear for the rules (LS- Global), (LS- Fut), (LS- Empty), (LS- Delete) and (LS- Plus). The result for the rule (LS- Bind) is a consequence of (Red- AInvk). The result for the rule (LS- AInvk) is a consequence of the definition of $\Rightarrow$. The result for the rule (LS- SInvk) is a consequence of the definition of $\Rightarrow$ and (Red- SInvk). The result for the rule (LS- RSInvk) is a consequence of the definition of $\Rightarrow$ and (Red- RSInvk). Finally, the result for the rule (LS- DepNull) is a consequence of the definition of $\Rightarrow$.

Item 3 is obvious because circularities are preserved by injective renamings of cog names. □

References

1. Abadi, M., Flanagan, C., Freund, S.N.: Types for safe locking: static race detection for java. ACM Trans. Program. Lang. Syst. **28**, 207–255 (2006)
2. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe program: preventing data races and deadlocks. In: Proceedings of OOPSLA '02, pp. 211–230. ACM, London (2002)
3. Carlsson, R., Millroth, H.: On cyclic process dependencies and the verification of absence of deadlocks in reactive systems (1997)
4. Caromel, D.: Towards a method of object-oriented concurrent programming. Commun. ACM **36**(9), 90–102 (1993)
5. Caromel, D., Henrio, L., Serpette, B.P.: Asynchronous and deterministic objects. In: Proceedings of POPL'04, pp. 123–134. ACM, London (2004)
6. Comtet, L.: Advanced Combinatorics: The Art of Finite and Infinite Expansions. Reidel, Dordrecht (1974)
7. Coppo, M.: Type inference with recursive type equations. In: Proceedings of FoSSaCS, LNCS, vol. 2030, pp. 184–198. Springer, Berlin (2001)
8. de Boer, F., Bravetti, M., Grabe, I., Lee, M., Steffen, M., Zavattaro, G.: A petri net based analysis of deadlocks for active objects and futures. In: Proceedings of Formal Aspects of Component Software—9th International Workshop, FACS 2012, Lecture Notes in Computer Science, vol. 7684, pp. 110–127. Springer, Berlin (2012)
9. de Boer, F., Clarke, D., Johnsen, E.: A complete guide to the future. In: Programming Language and Systems, LNCS, vol. 4421, pp. 316–330. Springer, Berlin (2007)
10. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. SIGPLAN Not. **37**(5), 234–245 (2002)
11. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: PLDI 03: Programming Language Design and Implementation, pp. 338–349. ACM, London (2003)
12. Flores-Montoya, A., Albert, E., Genaim, S.: May-happen-in-parallel based deadlock analysis for concurrent objects. In: Proceedings of FORTE/FMOODS 2013, Lecture Notes in Computer Science, vol. 7892, pp. 273–288. Springer, Berlin (2013)
13. Gay, S., Hole, M.: Subtyping for session types in the π-calculus. Acta Inf. **42**(2–3), 191–225 (2005)
14. Giachino, E., Grazia, C.A., Laneve, C., Lienhardt, M., Wong, P.Y.H.: Deadlock analysis of concurrent objects: theory and practice. In: iFM'13, LNCS, vol. 7940, pp. 394–411. Springer, Berlin (2013)
15. Giachino, E., Kobayashi, N., Laneve, C.: Deadlock detection of unbounded process networks. In: Proceedings of CONCUR 2014, LNCS, vol. 8704, pp. 63–77. Springer, Berlin (2014)

16. Giachino, E., Laneve, C.: Analysis of deadlocks in object groups. In: FMOODS/FORTE, Lecture Notes in Computer Science, vol. 6722, pp. 168–182. Springer, Berlin (2011)
17. Giachino, E., Laneve, C.: A beginner's guide to the deadLock Analysis Model. In: Trustworthy Global Computing—7th International Symposium, TGC 2012, Revised Selected Papers, Lecture Notes in Computer Science, vol. 8191, pp. 49–63. Springer, Berlin (2013)
18. Giachino, E., Laneve, C.: Deadlock detection in linear recursive programs. In: Proceedings of SFM-14:ESM, LNCS, vol. 8483, pp. 26–64. Springer, Berlin (2014)
19. Giachino, E., Lascu, T.A.: Lock analysis for an asynchronous object calculus. In: Proceedings of 13th ICTCS (2012)
20. Henglein, F.: Type inference with polymorphic recursion. ACM Trans. Program. Lang. Syst. **15**(2), 253–289 (1993)
21. Igarashi, A., Kobayashi, N.: A generic type system for the pi-calculus. Theor. Comput. Sci. **311**(1–3), 121–163 (2004)
22. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Aichernig, B., de Boer, F.S., Bonsangue, M.M. (eds.) Proceedings of 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010), LNCS, vol. 6957, pp. 142–164. Springer, Berlin (2011)
23. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Softw. Syst. Model. **6**(1), 39–58 (2007)
24. Kerfoot, E., McKeever, S., Torshizi, F.: Deadlock freedom through object ownership. In: T. Wrigstad (ed.) 5rd International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO), in conjunction with ECOOP 2009 (2009)
25. Kobayashi, N.: A partially deadlock-free typed process calculus. TOPLAS **20**(2), 436–482 (1998)
26. Kobayashi, N.: A new type system for deadlock-free processes. In: Proceedings of CONCUR 2006, LNCS, vol. 4137, pp. 233–247. Springer, Berlin (2006)
27. Kobayashi, N.: TyPiCal (2007). http://www.kb.ecei.tohoku.ac.jp/~koba/typical/
28. Laneve, C., Padovani, L.: The must preorder revisited. In: Proceedings of CONCUR 2007, LNCS, vol. 4703, pp. 212–225. Springer, Berlin (2007)
29. Milner, R.: A Calculus of Communicating Systems. Springer, Berlin (1982)
30. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, ii. Inf. Comput. **100**, 41–77 (1992)
31. Naik, M., Park, C.S., Sen, K., Gay, D.: Effective static deadlock detection. In: IEEE 31st International Conference on Software Engineering, 2009. ICSE 2009, pp. 386–396 (2009)
32. Pun, K.I.: behavioural static analysis for deadlock detection. Ph.D. thesis, Faculty olf Mathematics and Natural Sciences, University of Oslo, Norway (2013)
33. Puntigam, F., Peter, C.: Types for active objects with static deadlock prevention. Fundam. Inf. **48**(4), 315–341 (2001)
34. Requirement elicitation (2009). Deliverable 5.1 of project FP7-231620 (HATS). http://www.hats-project.eu/sites/default/files/Deliverable51_rev2.pdf
35. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)
36. Vasconcelos, V.T., Martins, F., Cogumbreiro, T.: Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In: Proceedings of PLACES'09, EPTCS, vol. 17, pp. 95–109 (2009)
37. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. **10**(2), 203–232 (2003)
38. West, S., Nanz, S., Meyer, B.: A modular scheme for deadlock prevention in an object-oriented programming model. In: ICFEM, pp. 597–612 (2010)
39. Wong, P.Y.H., Albert, E., Muschevici, R., Proença, J., Schäfer, J., Schlatte, R.: The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. J. Softw. Tools Technol. Transf. **14**(5), 567–588 (2012)
40. Yonezawa, A., Briot, J.P., Shibayama, E.: Object-oriented concurrent programming in ABCL/1. In: Proceedings of OOPSLA'86, pp. 258–268 (1986)

Elena Giachino is Research Assistant at the Department of Computer Science and Engineering of the University of Bologna. She has been involved in several European Projects. Her research interests cover the area of type systems for concurrent /functional/object-oriented languages, the analysis of program properties and reversible computations.

Cosimo Laneve (Ph.D., 1993) is Professor of Computer Science at the University of Bologna since 2002. He has been PC member/invited speaker at international conferences and workshops, currently member of IFIP WG 1.8. He has also been involved as principal investigator in several EU, Microsoft and national projects. His research interests cover the area of the theory of programming languages, in particular, concurrent/functional/object-oriented languages, their implementation, and the analysis of program properties mostly using behavioural types.

Michael Lienhardt is a PostDoc at the Department of Computer Science and Engineering of the University of Bologna. He has been involved in several European Projects. His research interests cover the area of analysis of concurrent/functional/object-oriented languages using tools like type systems, and the design of such programming languages using paradigms like component models or reversible computation. He also likes to put his formal research in practice by providing a prototype implementation of his work.